Course Script

IN 5110: Specification and Verification of Parallel Systems

IN5110, autumn 2019

Martin Steffen, Volker Stolz

Contents

5 Partial-order reduction **1**
5.1 Introduction 1
5.2 Independence and invisibility 9
5.3 POR for LTL$_{-\bigcirc}$ 12
5.3.1 Calculating the ample sets 24

Chapter 5
Partial-order reduction

What is it about?

Learning Targets of this Chapter

The chapter gives an introduction to *partial order reduction*, an important optimization technique to avoid or at least mitigate the state-space explosion problem.

Contents

5.1 Introduction 1
5.2 Independence and invisibility 9
5.3 POR for LTL$_{-\bigcirc}$ 12

5.1 Introduction

The material here is based on chapter 10 from the book [1] or the handbook article [2].

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*

All analyses based on "exploration" or "searching" suffer from the fact that problems become unmanagable when confronted with realistic problems. That's also true for other approaches like SAT/SMT (but there is not lack intractable problems in all kinds of fields). If we look as (explicit-state) model checking very naively, and perhaps even focus on only very simple problems like checking $\Box\varphi$ ("always φ"), the model checking problem phrased like this seems not really untractable (complexity-wise). It's nothing else than graph search, checking "reachability" of a state that violates the property φ. Searching through a graph is *tractable* (it has *linear* complexity, measured in the size of the graph, i.e., linear in the number of nodes and edges). So that's far from "untractable".

In model checking, it's the size of the graph, that causes problems. That's typically exponential in the description of the program. In model checking one is often interested in temporal properties of reactive, concurrent programs, consisting of more than one process or thread running in parallel. Typically, the size of the global transition system "explodes" when increasing the number of processes, due to the many interleavings of the different local process behaviours one needs to explore.

Of course, given a program or model there may be other sources that makes a raw state exploration unmanagable. If the problem depends in data input (like inputting numbers), the the size of the problem increases exponentially with the "size" of the input data. If one integers with only one byte length, one already has to take 2^8 inputs into account. Normally, of course, one has (immensely) larger data to deal with, and perhaps not just with one input, but repeated input in a reactive system. Those kind of data dependence quickly goes out of hand. It's also a form of "state explosion problem", but mostly, when talking about the state-explosion problem for model checking, one means the state explosion due to different interleavings of concurrent processes. Dealing with data is not the strong suit of traditional model checkers, so it sometime better to deal with data with different techniques, and/or to ignore the data. This means to abstract away from data (that's known as data abstraction) and let the model checker focus on the part of the problem it is better suited, the reactive behavior and temporal properties.

Especially, partial-order reduction covered in this section is a technique, specially made to work well to reduce irrelevant interleavings.

One last word about "complexity": Before we said that model checking is linear in the size of the transition system. That's of course an oversimplification, insofar that the "size" of the formula plays a role as well. For instance, in the section about the μ-calculus it was hinted at the the alternation-depth is connected to the complexity of the model checking problem in that contaxt. For model checking LTL, the time complexity is actually exponential in the size of the formula. Normally, that's not referred to as state explosion and also in practice, the size of the formula is not then limiting factor (and if one has many problems to scheck, which can be seen as a big conjuctions), one can check the individual properties one by one.

Battling the state space explosion

Since it's such major road block, it's clear that many different techniques have be proposed, investigated, and implemented to address it. The list includes some major ones, of course also clever implementation and data representations and other programming-related techniques are used.

- symbolic techniques
- BDDs
- abstraction
- compositional approaches
- symmetry reduction
- special data representations
- "compiler optimizations": slicing, live variable analysis . . .
- parallelization
- here: *partial order reduction*

"Asynchronous" systems and interleaving

- remember: synchronous and asynchronous product (in connection with LTL model checking)

- asynchronous: softwared and asynchonous HW
- synchronous: often HW, global clock
- **interleaving** (of steps, actions, transitions ...)

Partial order reduction is most effective in asynchonous systems. The distinction is for systems with different parts working in parallel or concurrently, and one can make that distinction for HW or software. In HW, synchronous behavior can be achieved by a global HW clock, that forces different components to work in lock step. The global clock is used to *synchronize* the parts. Also in software, synchronous behavior has it's place (one could have protocols simulating or realizing a global clock) there are also so-called *synchronous languages*, programming languages based on a synchronous execution model, they are often used to model and describe HW, resp. software running on top of synchronous HW.

Concurrent software and programs, though, more typically behave *asynchronously*, i.e., without assuming a global clock. A good illustration are different independent processes inside an operating system, say on a single processor. The operation system juggles the different processes via a *scheduler*. The scheduler allocates "time slices" to processes, letting a process run for a while, until it's the turn of another process. In a mono-processor, it's one process at a time, and the scheduler *interleaves* the steps of a different processes. That's a prototypical asynchronous picture.

Of course, often processes or threads etc. don't run in a completely independent or "a-synchronous" manner. To allow coordination and communication (and perhaps to help the scheduler), there are different ways of *synchronization* and constructs for synchronization purposes (locks, fences, semaphores, barriers, channels ...). Very abstractly, synchronization just means to *restrict* the completely free and independent execution. Even if processes coordinate their actions using various means of synchronization, one still speaks of *asynchronous* parallelism. If one would go so far in tie the processes together by using by a sequence of global barriers, where each processes takes part in, then that very restrictive mode of synchronization would effectively correspond to having a global clock and synchronous behavior.

The 2 ways of compose two automata "in parallel" reflected those two ends of the spectrum: completely asynchronous and completely synchronous. (The definition was done for Büchi-automata, but the specifics of (Büchi-)acceptence are an orthogonal issue that have to do with the specific "logical" needs we had for those automata (representing LTL). The synchronous-vs.-asynchronous composition is independent from those details.

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

The name of the technique seems to promise reductions based on "partial order". We'll see about the reductions of the state space later, but why "partial order"? A partial order (or partial order relation) is a binary relation which is *reflexive*, *transitive*, and *anti-symmetric* (we encountered it in connection with lattices when dealing with the μ-calculus).

What's the connection? The short story is maybe the following: exploring the state space involves exploring different interleavings of steps of different processes. Often that means one can do steps either in one order in one exploration, or in reversed order in a alternative exploration (and the whole trick will be to figure out situations when the exploration of the alternative order is not needed). One will not figure out precisely *all* situations where one can leave out alternatives, that would be too costly. So, one conservatively overapproximate it: when in doubt with the available information, better explore it.

It's of course not *always* the case that one can reorder steps into an alternative order. Steps within the same process might well be executed in the order written down; likewise, synchronization and communication may enforce that steps are done in one particular order or at least that they cannot be freely shuffled around (that's, in a way, the whole point of synchronization). Anyway, one may therefore see the actions or steps as *partially ordered*, at least when considering the behavior of the system as a whole. Focusing on one run or path, of course, presents one particular schedule and the steps in that run appear in a *linear* or *total order*. In one particular run, it's not represented, if two events are ordered by necessity (one is the cause of the other for instance) or whether the order is accidental.

That's the short story. Based on ideas as discussed, people proposed ways to describe concurrent behavior different from the *interleaving* picture, but based on *partial orders*. Those kind of styles of semantics are connected to *true concurrency* semantics, to distinguish them from "interleaving semantics" (which thereby could be called a "fake concurrency" semantics...). There is a point to it, though. Remember the informal discussion of asynchronous processes and interleaving, referring to scheduling processes on a single-core processor. There, clearly concurrency is an illusion maintained by the operating system's scheduler, that juggles the different processes so fast that, for the human, they appear to be concurrent, whereas "in reality", there is at most one process actually executed at a time. Two things being concurrent, in that picture, is just a different way of saying, they can occur in either order. True concurrency semantics takes a diffent point of view, seeing concurrency as something different from just unordered. As simple litmus test: A semantics that considers $a \parallel b$ as equivalent to $ab + ba$ is an interleaving semantics, if the two "systems" are different, it's a true concurrent interpretation (details may apply). For instance, Petri-nets is a quite old "true concurrency" model (they exist also in many flavors, and they are other true concurrency models as well). True concurrency models make use of partial orders (and perhaps other relations as well), but we don't go into true concurrency models.

Independent from the true-vs-"fake" concurrency question: there is a connection between partial order semantics and semantics based on arbitray interleavings. It's a known mathematical fact that every partial order can be linearized (i.e., turned into a total order), and more generally, that a partial order is equivalent to the set of all its linearizations. The first statement, that partial orders are linearizable, may be known from the 2000-course "algorithms and data structures". In that course, a straightforward solution to the

problem is presented known as “Dijkstra’s algorithm”.

POR here takes as starting point sets of executions or runs, which are linearizations. It does not take as a starting point a partial-order semantics (let alone a true concurrency semantics). While connections between partial orders and linearizations are easy, well-known, and hold generally, i.e., for all partial orders, they are more an inspiration than a technical basis for partial order reduction here. Nailing down a concrete partial order semantics for *concrete* situations in an asynchronous setting with specific synchronization constructs is not so easy. It’s much easier to specify what a program can do for the next step; that leads to an operational semantics which also specifies all possible runs of a program. *Implicitly,* that also contains all alternative runs, so one could say (based in the mentioned “math fact”) that somehow indirectly one may view it as described a “partial order” between the steps of the behavior of the program. But it’s, as said “implicit” and for the behaviors per program.. But it’s for from easy to start upfront with a partial-order based semantics for all programs.

POR reduction therefore is not based directly on an explicit partial order semantics. It does not even strive to reconstruct fully the underlying partial order that is hidden in the set of all interleavings of one given program. It does something more modest (but also more ambitious at the same time, as it has to be done during the model-checking run and has to be done efficiently). Perhaps POR is inspired by the connection between partial orders and possible linearizations and partial order semantics, but one can understand POR even simpler: it’s intuitively clear that, under some circumstances, it does not matter in which way steps are done and in other circumstances it does. POR tries to figure out when alternative orders don’t matter and avoids them. That needs to be done *while* running the “program”, i.e., running the model checker, and compromises need to be done. It needs to be still efficent. It also (and connected to that) needs to be “local”. One cannot first generate all runs, then filter out duplicates, and then model check the rest, or something. Instead, when doing the exploring the state space, during the model checker run, a local decision needs to be made, like “shall I explore the next candidate edge, or can I leave it out.” Of course, the criteron should not be trival like: I leave an edges if I know that I have seen the resulting state already. That’s ridiculous, I might as well follow the edge and then backtrack ofter discovering that I have seen the state already. Exploring one edge more and then checking is not worth the effort compared to making some fancy overhead to avoid that last step. One has to do better namely, one can leave out an edge, if all what follows is covered already or actually what will be covered later. At any rate, one cannot expect those estimations to be *precise* in recovering all of the theoretically possible reduction (if one had a full partial-order picture, which one does not have anyway).

The contrete details when to explore and edge and when not depend also on the language and its constructs. For instance, if one has shared variable, and the model checker is in a state where, in a next step, process 1 can write atomically to a variable or process 2 can write atomically to the same variable, it’s clear that one has to explore both alternatives. Or does one? What if they write the same value? Well, perhaps it’s not worth in checking that, one may conservativel explore both orderings anyway, perhaps it’s not worth the effort.

To postponed the details of more concrete language constructs for later, one abstract away a bit first and introduces the concepts of *dependence* and *independence* (for instance,

two reads to the same variable may be independent, whereas two writes may not). The theory justifying the POR is then based on notions of independence and when the order of execution is irrelevant and can be commuted. That is perhaps inspired by partial-order thinking, but can be an approximation at best (already for practical reasong), therefore a better name of partial-order reduction may be **commutativity-based reduction** (see [2] who makes that argument).

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

- see also *symmetry reduction*
- 8 queens problem
- POR: equivalent *behaviors*

(Labelled) transition systems

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

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

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 $s \xrightarrow{\alpha}$

Otherwise $\xrightarrow{\alpha}$ *disabled* in s.

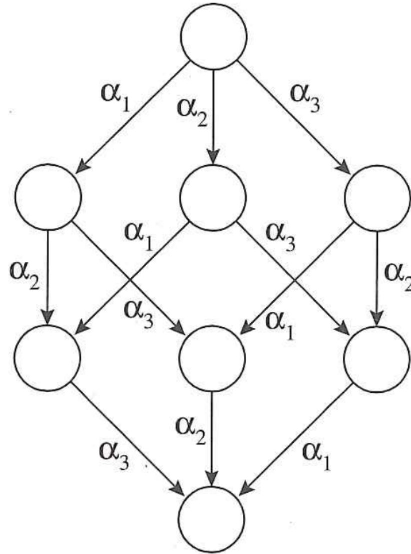
- path π:

$$s_0 \xrightarrow{\alpha_0} s_1 \xrightarrow{\alpha_1} s_2 \xrightarrow{\alpha_2} \ldots$$

- not necessarily infinite

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

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*

Modification/improvement Don't explore *all* enabled transitions.

follow **enough enabled** transition

- ample: think "sufficient" or "enough"
- **ample** set of transitions in a state $\subseteq$ set of enabled transitions in a state

Modified DFS

```
1   hash(s0);
2   set on_stack(s0);
3   expand_state(s0);

4   procedure expand_state(s)
5       work_set(s) := ample(s);
6       while work_set(s) is not empty do
7           let α ∈ work_set(s);
8           work_set(s) := work_set(s) \ {α};
9           s' := α(s);
10          if new(s') then
11              hash(s');
12              set on_stack(s');
13              expand_state(s');
14          end if;
15          create_edge(s, α, s');
16      end while;
17      set completed(s);
18  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*

- so far:
  - quite wishy-washy, only general idea
  - "unrealistic" (as mentioned)

- details also dependent on the "programming language"
- alternatives of *ample sets* with analogous ideas (the names are not really indicative of how all that works):
  - sleep sets
  - persistent sets
  - stubborn sets
  - ...

With a little help of the programmer ...

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

`atomic` atomic block executed indivisibly

`D_step` deterministic code fragment executed indivisibly.

Rest

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

5.2 Independence and invisibility

2 relations between relations

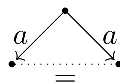
- we have labelled transitions (resp. multiple relations)
- 2 important conditions for POR
  - one connects *two relations*
  - one connects one relation with the property to verify

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

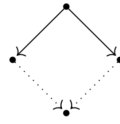
Invisible Taking a transition does not change the satisfaction of relevant formulas

Determinism, confluence, and commuting diamond property

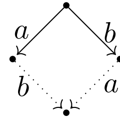
Determinism



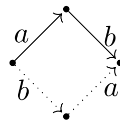
Diamond prop.



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 5.2.1 (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 $(\xrightarrow{\alpha_1}, \xrightarrow{\alpha_2}) \in I$

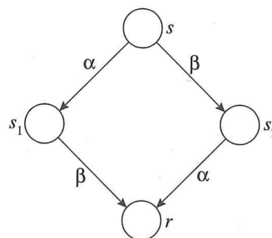
Enabledness If $\alpha_1, \alpha_2 \in enabled(s)$, then $\alpha_1 \in enabled(\alpha_2(s))$

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?



2 issues

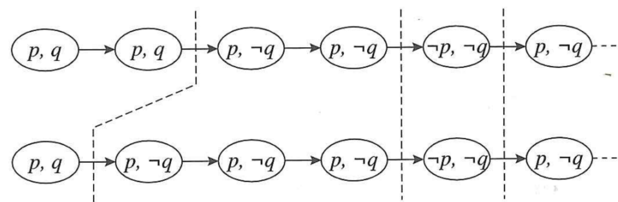
1. The checked **property** might be sensitive to the choice between s_1 and s_2 (and not just depend on s and r
2. s_1 and s_2 may have **other successors** not shown in the diagram.

Visibility

- $L : S \to 2^{AP}$
- $\stackrel{\alpha}{\to}$ is **invisible** wrt. to a set of $AP' \subseteq AP$ if for all $s_1 \stackrel{\alpha}{\to} s_2$

$$L(s_1) \cap AP' = L(s_2) \cap AP'$$

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
- $\mathrm{LTL}_{-\bigcirc}$: "next-free" fragment of LTL (often also $\mathrm{LTL}_{\text{-X}}$)

Stuttering

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

5.3 POR for $LTL_{-\bigcirc}$

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**:

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

- note: "iff"
- mainly a condition on *paths*

Path representatives each path π_1 in $\mathcal{M}$ starting in s is represented by an **equivalent** path π_2 in $\mathcal{M}^{\text{✂}}$, 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**). include 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.

The discussion is based on the presentation in [1].

C1 That's said to be the most complex condition. It seems to have been ediscussed (maybe even under different names) in the literature. It is furthermore stressed, that the condition explisitly refers to the *full* unreduced state graph and paths throughout those. Path here are alternating state-transition sequences. Note that unlike in "basic semantics" for LTL, the paths are also path *through* the transition system or graphs. It's important, insofar that the states in the graph are relevant, in particular also which other actions are alternatively enabled, resp. which alternative actions are in the ample sets at a given state. So, a path (or execution) not a fully linear structure, it's a bit like the picture of "barbed wire" in the definition of bisimulation, or of resusal sets etc.

Anyway, the condition C1 states

> Along *every* path in the *full* state graph that starts at S, the following condition holds: a transition that is dependent on a transition in $\mathit{ample}(s)$ cannot be executed without a transition in $\mathit{ample}(s)$ occurring first.

The choice of word "depending on" is a bit unfortunate. Dependence, which is the opposite of independence, is *symmetric*. Maybe "interdependence" would be a better word. Also the status of "events" are unclear, it's all quite abstract, but independence means: not disablbling, therefore depenence can mean disabling (but there's no talk about "enabling"). Let's postpone that a bit.

Note that the condition also does not speak about *individual* transitions. It talks about "some" transitions. Timewise, there are 3 points in time relevant: now, in state s, in the future, and in between those 2 timepoints. But remember: the condition talks about *all* paths starting in s, the condition and the 3 time points refer to all those.

As a consequence of the definition: in a state, all enabled ones but not ample are independent from the ample ones. That's indeed a direct consequence of the condition. If a transition (assuming otherwise) that is dependent on one in ample would both be enabled but not covered by the ample set, then one could simply start a path using that transition, contradicting the condition.

The definition of "ample" has two aspects. One is the next-step condition, that's the one use in the algorithm. Ample transitions are the prioritized ones, the one in the focus. For the next-step picture, it's black and white: there is a set of enabled transitions. The ample ones are explored, the rest not. We are talking here in the "unrealistic" setting that we have the unrestricted graph at hand and the ample set allows to restrict the DFS in the described manner. Choosing one element locally from the ample set may *disable* an alternative, so there may be a *choice* inherent at a given point. Of course, we are here not dealing with single *executions* or *runs*, but with model checking. So the DFS will come back eventually and explore relevant alternatives as well.

Now, that was the next-step perspective on the ample sets. There is also the temporal aspect, capture in condition C1: for *all* runs starting somewhere, the relevant ones takes some form of priority as well. The condition is a bit more tricky than just having the relevant ones in a state before the irrelevant ones. The reason is, that transitions can enable as well as disable other transitions. However, that's not universally true. Independent transitions cannot *disable* each other (enabling is allowed). The ample set (the relevant transitions) are closed under interdepenence. As stated above in the lemma, it's an easy consequence of the "definition" of the relevant and irrelevant next steps, that the two sets are independent.

It's straightforward to prove: *2 swappable and deterministic relations that don't disable each others satisfy the commuting diamond property.*

This statement forbid disabling. The requirement about prohibitin enabling is used if one wants to prove the reverse implication. But it does not seem to enter the picture here. That sounds also a bit dubious to call two relations independent if one may enable the other. It seems to be related to a subtle distinction on the formulation of "swappability" and the condition of commutativity as part of the independence here. The requirement corrending to "no-enabling" is covered by the fact that commutativity in the formulation of [1] requires that the to relations involved for which the commutation holds are both enabled.

Now, coming back to the definition of condition C1. The ultimate goal which one hopes co capture with C1 is of course correctness of the modified DFS, i.e., one does not miss out relevant paths.

The argument looks at path in the unreduced graphs (which is what C1 talks about, and likewise in principle also C0, except that it does not even talk paths, just about states, so in that perspective there is no difference between the reduced graph and the non-reduced one). Alright. If we taken an arbitrary path in the non-reduced graph, starting at a point, one is not forced to start with a relevant transition (from the ample set). That's what the reduced graph would do. The liberal behavior can do things outside the ample set. Of course, we should be aware that that it's not "the" ample set, the one where we start the path. After doing a step, we are in a different state, and that state may have a different ample set (and a different set of low-priority transitions, since the set of enabled transitions can surely change). However, by taking an irrelevant β, we know that it's independent (as required by C1) and therefore it cannot disable the relevant ones. In the successor states, the situation is new, but the set of relevant transitions cannot get smaller. They may be other irrelevant transitions, but again according to C1, those (perhaps different) transitions are likewise subject to the independence requirement, now in the changed state. Still, the cannot disable the "older" or "inherited" transitions, so they keep on lingering, as long as one takes non-relevant transitions. One can also not discard them by running into a dead end. That's forbidden by C0. That means, for an element α of the relevent transitions in a state. In a path starting there, there will finitely many βs followed by an α, or there are infinitely many βs. Again, the definition is semi-fixed on the state s with respect to the αs, but the βs are not defined relative to s.

Anyway, assume we are at state s and look at paths starting there. Now, the path can have two forms, either it has a prefix $\vec{\beta}\alpha$ or it's infinite $\vec{\beta}$..., where α is in the ample set of s and the β_i are all independent for all transitions in the ample set of s (which includes α). Note that at this particular point the text does not require that the β are outside $ample(s)$ (which would also a weaker requirement than requiring that the actions are from $enabled(s) \setminus ample(s)$, of course). At that point in the text, the only requirement is that the β are independent from $ample(s)$ (but $ample(s)$ may contain elements which are among them selves independent, that means, α may actually *not* be the first element from $ample(s)$ that is being chosen in the first sequence). Similar remarks apply to the second form of an infinite β-sequence.

The text may not require that, but either intuitively it's assumed (because that's the interesting case), resp. that additional assumption is added in the further discussion. Maybe they are just slightly sloppy. Anyway, without adding the additional requirement, the sequence $\vec{\beta}\alpha$ may actionally be taken by the modified DFS exploration (it's only not interesting). Interesting is only the case where the βs are all *not* from $ample(s)$.

In that case where α is the *first* element from $ample(s)$, the sequence $\beta_0, \ldots, \beta_m\alpha$ is indeed *not* chosen by the modified DFS (in case $m > 0$, which is also not mentioned in the text).

So, let's refer to sequences written $\vec{\beta}$ and starting in s containing *no* members of the relevant set $ample(s)$.

The formulation in the book is a bit shifting with the

The sequences of the form $\vec{\beta}\alpha$ may *not* be taken by the modified DFS. Technically, the book here is imprecise, it states that sequences of that form *are* in fact not taken. That seems not correct. It's not an real error. The argument that follows states that instead of this sequence, another one is taken. That's true indepedent from the question wether the original sequence $\vec{\beta}\alpha$ is taken by the algo or not. Therefore one could say, the given argument covers only the interesting case where the algorithm does not take that sequence. The text also assumes that none of the elements of $ample(s)$ are taken in $\vec{\beta}$. Also that does

$\bowtie$

#

cycle condition Willems and Wolper [3]

1. Choice, conflict, and interdependence
   Choices are covered, I think.
2. Transitions, relations, and events

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 $\mathcal{M}$ 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*.

- easy fact: $ample(s) \bowtie \neg ample(s)$

Form of paths in $\mathcal{M}^{\text{✂}}$

- consequence of $\mathbf{C}_1$: two forms of paths

Blocks

1. with prefix $\beta_0\beta_1 \ldots \beta_m\alpha$
   - $\alpha \in ample(s)$
   - $\beta_i \bowtie ample(s)$
2. without such prefix:
   - infinite $\beta_0\beta_1\beta_2 \ldots$
   - $\beta_i \bowtie ample(s)$

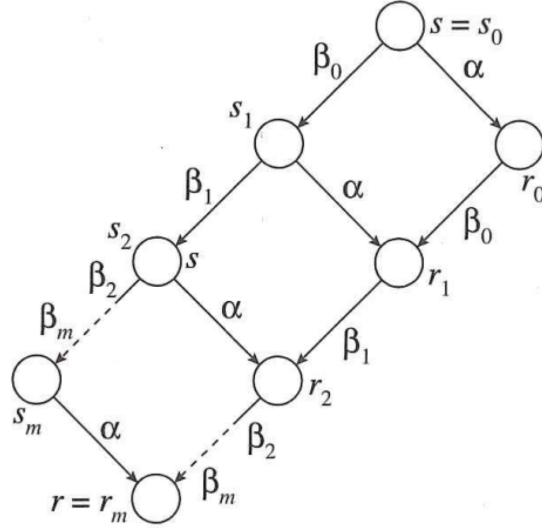
- assume: all $\beta_i \notin ample(s)$
- same as $\beta_i \in \neg ample(s)$?

Commutation

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

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

1. Pic



2. Paths
   - $\pi_1 = \vec{\beta}\alpha$
   - $\pi_2 = \alpha\vec{\beta}$

- $\pi_1 \in \mathcal{M}$ implies $\pi_2 \in \mathcal{M}$ (and vice versa)
- **what about $\mathcal{M}^{\text{✂}}$?**: $\pi_1 \notin \mathcal{M}^{\text{✂}}$ $(m > 0)$ and $\pi_2 \in \mathcal{M}^{\text{✂}}$

Explanations The assumptions of *independence* means that, in the original transition system $\mathcal{M}$ the following holds: if (starting in s) path π_1 is possible, then so is π_2, both ending in the same end state. The reason is that part of the condition of independence is that actions can be swapped or commuted. So, as far as their *existance* in $\mathcal{M}$ is concerned, π_1 and π_2 are "equivalent" (and all the "intermediate" paths as well, like $\beta'\alpha\beta''$).

In the pruned system $\mathcal{M}^{\text{✂}}$, things change. In particular, the "upper" path π_1 which puts α at the end, does not exist (in case $m > 0$): we assumed that in particular, $\beta_0 \notin ample(s)$, so already the first step is not possible.

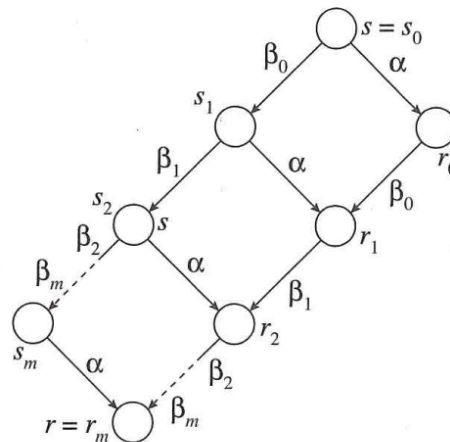
Now, as said, both paths are interchangable wrt. their existance: if one path exists, it's guaranteed that the other exists, and vice versa and they have the same start and end state (s and r in the picture). But are they interchangable also wrt. to the intermediate, visisited state, in particular, are the two paths interchangeble wrt. the property we model check? Well, one paths visits $s_0, s_1, \ldots s_m, r$ the other one $s, r_0, \ldots, r_m$ (with start and end states coinciding, i.e., $s_0 = s$ and $r_m = r$). So the question is: does it matter if one passes though the states r_i or the states s_i?

Of course, it may matter if some property holds for r_i but not for s_i or vice versa. The r_i and s_i states are connected by α, i.e.

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

Now, whether π_1 or π_2 is taken (or one of the "intermediate mixtures) does not matter provided that same formulas hold, comparing r_i with s_i. That's guaranteed if α is invisible (with respect to the atomic propositions)

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?

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

Explanation The answer is clearly *no, it does not matter* provided that the satifaction or "dissatisfaction" of the property does not depend on whether one is in s_i or r_i. That form of "invariance" has been called "invisibility". The perspective is that the a formula *observes* the transition system, it can "see" if a truth status changes (from true to false or the other way around). Observing *changes* means being able to observe transitions. And, in this picture, a transition is invisible or not observable, if taking said transition doe not lead to change of any truth values. Actually, visibility has been defined with resp. to atomic propositions only, more complex formulas don't need to be considered, resp. their non-observability follow as a consequence.

Invisibility of transitions

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

C_2 (invisibility) If s is not fully expanded, then every $\alpha \in \mathit{ample}(s)$ is *invisible*.

Partial order reduction allows to ignore steps. In the way it's presented here, locally, per state, the (DFS) exploration focuses "ample sets" of the enabled next steps, and omits the rest. That's interesting only if really some elements are ignored. As shown in the above example, we have to be careful when ignoring transitions (for instance the tsb_0).

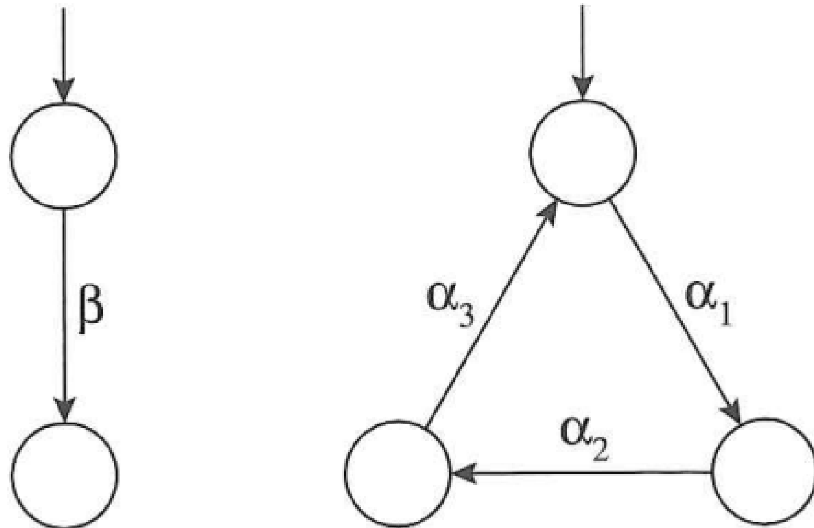
> If we ignore transitions, the non-ingored transitions must all be *invisible*.

As for terminology on the slide: A state s is **fully expanded** if $\mathit{ample}(s) = \mathit{enabled}(s)$. That's a situation where all enabled transitions are explored anyway, so in that case, the ample-set at s is certainly ok, without need to require invisibility of any transitions.

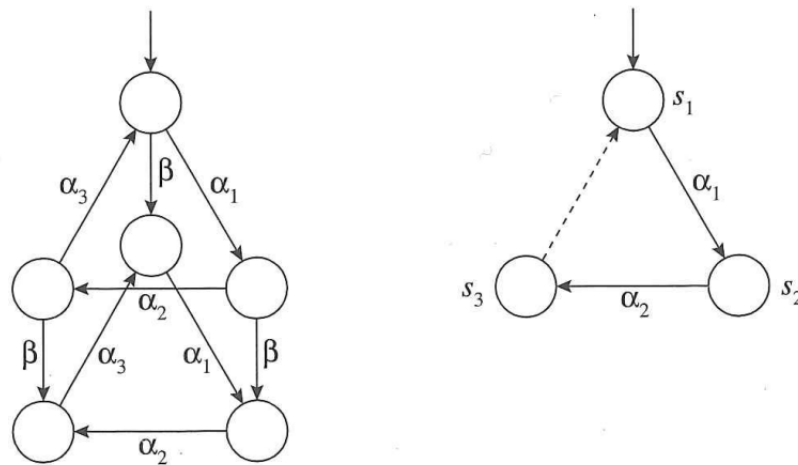
Is that all?

Pics

1. Two concurrent procs



2. $\mathcal{M}$ and $\mathcal{M}^{\times}$



Is that all?

The previous condition insisted on invisiblity of an action α, in case one omits alternatives. The picture shown previously illustrated that, that if α is invisble, the uncovered path (in the picture) with α at the end can be reordered with α at the beginning *without* omitting intermediate states with different logical status. That last condition about invisibility took care about *one* form of paths that are required as consequence of condition $\mathbf{C}_1$, namely the one with finitely many β_i's (not in the ample set of a state s) followed by one α from the ample set of s.

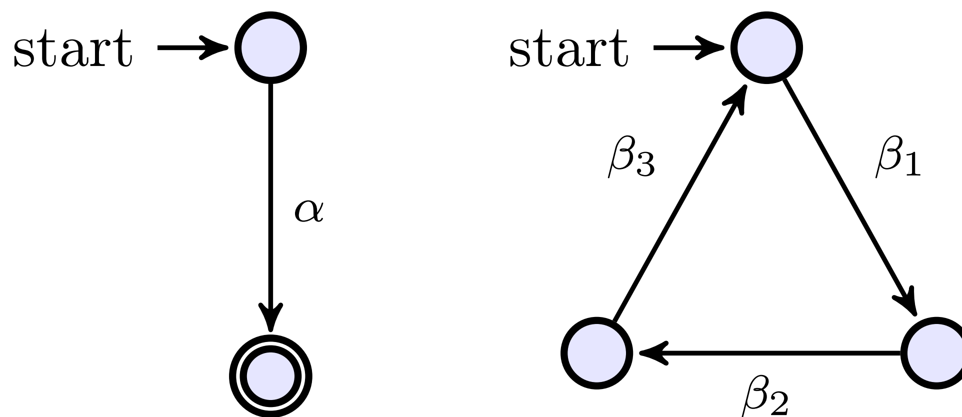
The final condition $\mathbf{C}_3$ needed for the correctness of pruning the exploration to focus on the ample-sets has to do with the second form of paths that follow as consequence of $\mathbf{C}_4$, namely the ones with infinitely many β_i, and *never* any α from the ample set.

Based in the intution of the ample sets we can already intuitively see that one has to be careful there. The ample set in a state represent the transitions that should be explored, and the rest from $\neg ample(s)$ are the one that intended to be ignored (because one can argue that they are equivalently covered otherwise during the exploration). Now, a transition in the ample set of a state marks it as "the transition needs to be explored". Postponing it "forever" is not the way to go.

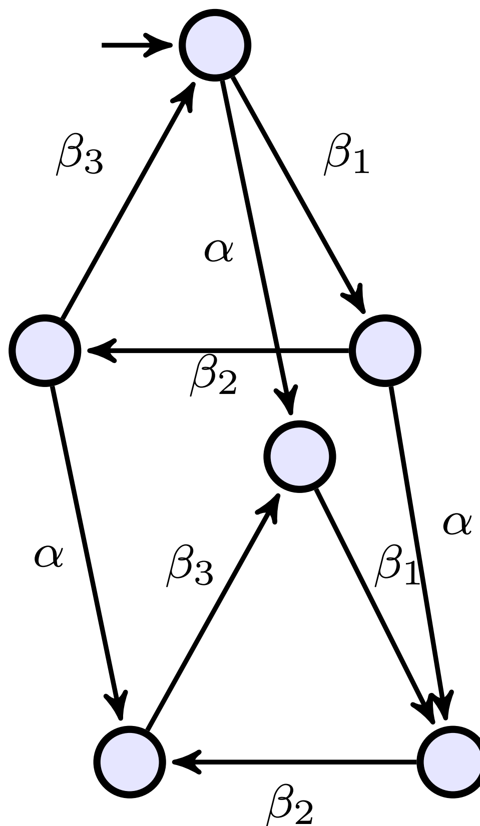
The condition $\mathbf{C}_3$ (like the other conditions) is not a condition on the behavor or the form of paths (like "don't look at paths where transitions $\alpha \in ample(s)$ are postponed forever"), it's a general condition on the forms of the ample set in the state that must be designed in such a way that, when running the system, all paths have the desired properties (in particular guaranteeing correctness, or avoiding infinite postponements of the form sketched).

Pics

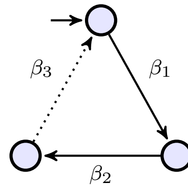
1. Two concurrent procs



2. $\mathcal{M}$



$\mathcal{M}^{\times}$



The three figures serve to illustrate the previously discussed problem of "infinite post-ponement". To comple the example, we need to add one piece of information, namely the "logical part", i.e., at which states satisfy which propositions.

We make it very simple; the goal is

- α is **visible**
- the β_is are **invisible** (the second process stuttering)

Intuitively it can be achieved by assuming that there is one boolean variable, initially say "false", and the process to the left sets it to "true" via it's transition $\xrightarrow{a}$. The "label" a may represent the assignment `p := true`. The other process does not do anything (except spinning around, cycling through its three states).

The first pc show two processes running in parallel in an asynchrous fashion, i.e., inter-leaving their steps. The overall combined behavior is given by the transition system $\mathcal{M}$, with 6 states. In that $\mathcal{M}$ with its 6 states and if we assume one propositional atom p, then p is false in all 3 states on the top of the picture, and true in the three states on the bottom.

For this system, we can find ample sets that satisfy all the three conditions so far, but still fail to achieve correctness. That's easily doable by *systematically ignoring* α, i.e., not including this transition in any of the ample sets. I.e., each state has an one element ample set $ample(s) = \{\beta_i\}$, and α is not included anywhere.

It's easy to check that this choice satisfies $\mathbf{C}_0$ (trivally, since no ample set or enabled set is empty), $\mathbf{C}_1$ (since α is assumed to be independent from the β_is; remember that $\mathbf{C}_1$ speaks about paths in $\mathcal{M}$, not in $\mathcal{M}^{✂}$). And finally $\mathbf{C}_2$ is satisfied as well, as the example is constructed in such a way that the α_i are all **invisible**, as required by $\mathbf{C}_2$.

Transition α *is* visible (it does not stutter), so taking it matters wrt. the verification. However, the ample sets chosen as given, leads to explorations in $\mathcal{M}^{✂}$ ignoring α.

The last condition $\mathbf{C}_3$ on the next slide exclude such infinite avoidance. Seen as condition one the graph itself, it's a condition on a cycle (not a condition on infinite paths resp. only indirectly so, since in finite-state systems, ininite paths must come from running through at least one cycle). What needs to be ensure is that a situation as in the example cannot occur. That $\xrightarrow{a}$ is not included in *some* of the 3 states of the last picture is fine. What is not fine is that it's left out in *all* of them in the cycle. It would allow (as in the example) to construct a path running through this cycle where the transition is constantly enabled

but always in $\neg ample(s_i)$, so no state "takes responsiblity" to at least one time, explore that edge. In the example, the neglegted edge α is a **visible** one. But the requirement stating "do not systematically neglect an edge" also applies to invisible ones as well. Even if an edge is invible, one may reach behavior after taking it that *is visible* and needs to be checked. The example is also specific insofar in that $\xrightarrow{\alpha}$ is *continuously* enabled (but not taken). Condition $\mathbf{C}_3$ is more stringent: don't neglect a transition $\xrightarrow{\alpha}$, what is somewhere enabled in a cycle.

This condition is connected with the notion of *fairness*. It's a notion that is relevant in concurrent systems. In practical systems (like operating systems), it also can be understood as a property of a *scheduler*. In our example, with two processes, a behavior that constantly schedules the second process, with systematically ignoring the first one (despite the fact that it *could* do a step, namely $\xrightarrow{\alpha}$), that's a non-fair behair. Of course, after the first process has done $\xrightarrow{\alpha}$, it cannot do any further (no transition is enabled, and that will remain so as well, as the process is terminated). If, in that situation, the scheduler "choses" only $\xrightarrow{\beta_i}$ steps from the second process, but no steps from the first, that does not count as being unfair.

There are, though, two variations of the concept of fairness, namely *strong fairness* and *weak fairness*. The illustrating example corresponds to the *weak* variant (resp. it illustrates behavior which not weakly fear). Since it's not even weakly fair, it also fails to be strongly fair, though. It illustrates a situation, where $\xrightarrow{\alpha}$ is neglected despite being *constantly enable*. The chose infiniten path $\beta_1\beta_2\beta_3\beta_1\ldots$ has an inifinite sequence of points where α is *constantly* enabled. Weak fairness requires that one cannot have an action (like α) enabled infinitely long without also taking it. fairness

Strong fairness say: of an action is enabled *infinitely often* (but can be disabled in between the places when it's enabled again), then, for fairness sake, it must be taken: strong fairness means, if an action is enabled infinitely often in an execution, it needs also to be taken infinitely often.

Condition $\mathbf{C}_3$ coming up next corresponds to the strong variant of fairness.

A final side remark (to to relevant perhaps for POR): as part of the illustration example, the chosen β_i transitions are all invisible. The resulting behavior (without imposing $\mathbf{C}_3$) is not just unfair in the described sense, neglecting $\xrightarrow{\alpha}$, the behavior is also doing an infinite amout of do-nothing steps (here formulated by having the $\xrightarrow{\alpha_i}$ as invisible). The have no influence on the satisfaction of formulas. More practically, one can see then as no-operation or skip steps (sometimes executing `NOP` steps, eating up processor cycles without doing anything) or do-nothing "stutter" steps added to the model (like we did in LTL).

Either way: infinitely many do-nothing or skip or stutter steps is seen as a simple and discrete form of so called Zeno-behavior. That's in honor of an old Greek philosopher Zeno of Elea, who is remembered for some speculative paradoxes (retold by Aristotle), often concerning infitely many (smaller and smaller time) steps. The most well-known of those is probably the tale of Achilles and the tartoise, racing against each other.

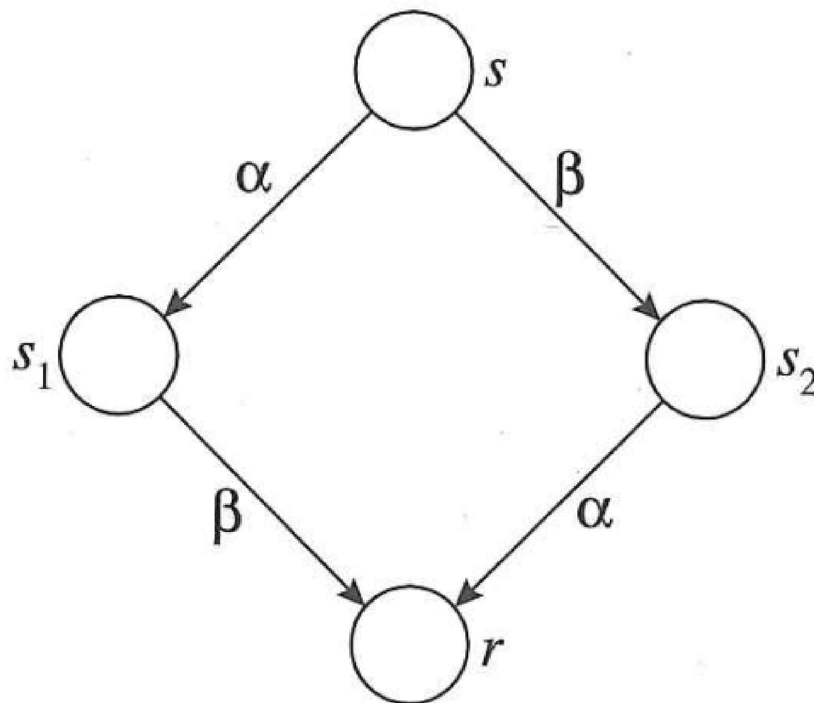
Cycle condition C_3

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

Repetition

1. Illustration

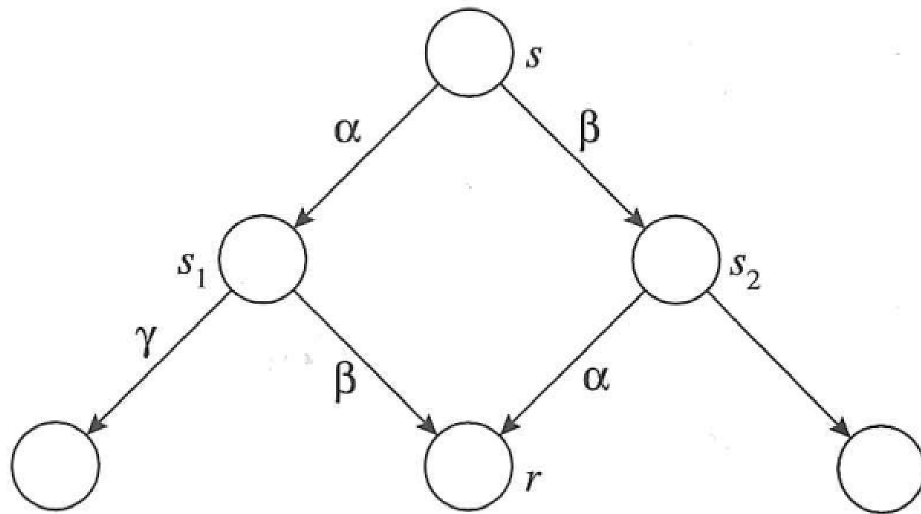


2. Text
   a) satisfaction depends in chosing path via s_1 or s_2?
   b) forgotten successors?

Rest

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

1. issue 2



2. the conditions imply
   a) $ss_2r \sim_{st} ss_1r$
   b) $ss_1s_1' \sim_{st} ss_2rr'$

5.3.1 Calculating the ample sets

Complexity

- checking conditions on-the-fly
- $\mathbf{C}_0$: easy
- $\mathbf{C}_1$: tricky
  - refers to $\mathcal{M}$, not $\mathcal{M}^{\text{✂}}$
  - 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

Rest

- 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 prococessses 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

```
1  function ample (s) =
2   for all P_i such that T_i(s) ≠ ∅ // try to focus on one P_i
3     if
4           check_C1(s, P_1) ∧
5           check_C2(T_i(s)) ∧
6           check_C3'(s, T_i(s))
7     then
8           return T_i(s)
9     if
10  end for all      // too bad, cannot focus on any but
11  return enabled(s) // fully expanded can't be wrong
12 end
```

Check C_2

```
1  function check_C2(X) =
2    for all α ∈ X
3    do  if   visible(α)
4        then false
5        else true
```

Check C_3'

```
1  function check_C3' (s,X) =
2    for all α ∈ X
3     do
4        if   on_stack(α(s))
5        then false
6        else true
```

Check C_1

```
1  function check_C1 (s, P_i) =
2    for all P_j ≠ P_i
3     do
4        if          dep(T_i(s)) ∩ T_j ≠ ∅
5           ∨
6                    pre(current_i(s) \ T_i(s)) ∩ T_j ≠ ∅
7        then return false
8     end forall;
9    return true
```

Bibliography

Bibliography

[1] Clarke, E. M., Grumberg, O., and Peled, D. (1999). *Model Checking*. MIT Press.

[2] Peled, D. (2018). Partial-order reduction. In Clarke, E. C., Henzinger, T. A., Veith, H., and Bloem, R., editors, *Handbook of Model Checking*. Springer Verlag.

[3] Willems, B. and Wolper, P. (1996). Partial-order methods for model checking: From linear time to branching time. In *Proceedings of LICS '96*, pages 294–303. IEEE, Computer Society Press.

Index

depth-first search, 7