Course Script

IN 5110: Specification and Verification of Parallel Systems

IN5110, autumn 2021

Martin Steffen

Contents

Chapter 6
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

6.1 Introduction

The material here is based on Chapter 10 from the book [2] or the handbook article [5]. [1] does not cover partial-order reduction.

A fundamental limitation in model checking is the *state-space explosion problem.* Model checking is basically intractable, i.e. it suffers from a combinatorial explosion which renders the state space exponential in the problem or system size.

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-solving, and there is no lack of intractable problems in all kinds of fields. If we look at (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 and as such 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, there may be other sources that make a raw state exploration unmanagable. If the problem depends on data input (like inputting numbers), the the size of the problem increases exponentially with the "size" of the input data. If one uses integers with only one byte length, one already has to take 2^8 inputs into account. Normally, of course, one has (immensly) 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. That is also a form of "state space explosion problem", but mostly, when talking about the state-space explosion problem for model checking, one means the state space 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 also 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.

Partial-order reduction is a technique to reduce the explored state space by avoiding 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 space explosion and also in practice, the size of the formula is not the limiting factor. Also, if one has many properties to check, which can be seen as a big conjuction, one can check the individual properties one by one.

Battling the state space explosion

As it's such major road block, it's clear that many different techniques have be proposed, investigated, and implemented to address it. An incomplete and somewhat unordered list symbolic techniques, BDDs, abstraction, compositional approaches, symmetry reduction, special data representations, parallelization of model checking, the use of "compiler optimizations" on the model (like slicing, live variable analysis . . .). And here we are doing *partial order reduction.*

"Asynchronous" systems and interleaving 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 hard- or software. In HW, synchronous behavior can be achieved by a global hardware clock, that forces different components to work in lock step. The global clock is used to *synchronize* the different parts. Also in software, synchronous behavior has its 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 (preemptive scheduling). In a mono-processor, it's one process at a time, and the scheduler **interleaves** the steps of 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 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 two 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? 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*

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, and 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 concurrency interpretation (details may apply).

For instance, Petri-nets is a quite old "true concurrency" model (they exist also in many flavors, and there 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 arbitrary 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-level 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 or 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 on the mentioned "math fact") that somehow indirectly one may view it as that it describes 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 far from easy to start upfront with a partial-order based semantics for all programs.

POR 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:

> 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".

Since this is done while running the model checker, compromises need to be done how much effort is done to estimate or predict if a step is necessary or not, as it needs to be efficient. 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. Instead, when 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 let it be."

Of course, the criterion should not be trival like: I leave out an edge if I know that I have seen the resulting state already. That's ridiculous, as one might as well follow the edge and then backtrack after discovering that I have seen the state already. Exploring *one more* additional edge and then checking is probably easier compared to to make some fancy overhead to avoid that very 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 programming *language* and its constructs. For instance, if one has shared variables, 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 checking that particular situation is not worth in checking, and one may conservatively explore both orderings anyway.

To postpone the details of more concrete language constructs for later, one abstracts away from concrete types of actions 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 (for practical reasons), therefore a better name of partial-order reduction may be **commutativity-based reduction** (see Peled [5] who makes that argument).

POR can be also understood as an example of analysis techniques that exploit *equivalences*

Exploiting “equivalences” means: Instead of checking all “situations”, figure which are **equivalent**. That also implies, equivalent wrt. the property being checked) and then check only one **representatives** (or at least not all) per equivalence class.

There are other such techniques along those line. One is known under the name *symmetry reduction*; we don’t cover that, but the underlying idea is simple. Often systems have some symmetries, one can exploit. Not that it’s a typical model-checking problem, but think of the 8 queens problem: is it possible to place 8 queens on chess-board without that they can attack each other in one move? If one wants to solve that combinatorially, one can exploit the symmetry inherent in the fact that a solution can be rotated 90 degrees and it’s remains a solution.

Of course, in this well-known example the symmetry is obvious, and someone who want to solve the puzzle may arrange a search algorithm (and/or the respresentation of the chess board) so make use of that knowledge. In general, symmetries may not be so obvious and one would like to have the model checker detect and exploit them, but that’s the general idea. And that is quite similar in POR, where the model check tries to detect equivalent *behaviors*.

(Labelled) transition systems In the lecture we have variously encountered representations based on states and transitions. Kripke structures or Kripke models, transition systems, and also automata belong into that category. There may be minor differences in terminology (states vs. worlds) and perpaps notation, but these representations are not radically different. Maybe the biggest difference is that we made use of the automata as an mechanism for *accepting* runs of a system, in particular the Büchi-automata used for LTL model checking accept infinite-word languages. Transition-systems, on the other hand, represent the system, program, or model under investigation, and do not have accepting states. Another difference was that automata were edge-labelled with letters from some alphabet, whereas for the transition systems we focused on information carried by the state. For instance, one could see propositional information to be a form of labelling: each state is “labelled” with the sets of propositional atoms that are assumed to hold in that state. Another terminology we also used was “valuation”.

At any rate, the edges of transition systems so far did not play an important role, but now we will be dealing with edge-labelled transition systems.[1] Typically, if one talks about *labelled* transition systems (LTS), one means transition systems with the *edges*-labelled (independent of wether or not the states are (also) labelled with propsotional information).

Now, the labels in the edges become important, because we need to look into when steps (i.e., transitions) in a system can be re-ordered resp. when some steps can be ignored in an exploration. To do so, the steps should carry information about the action they are representing, like: when a read from x by process P_1 is followed by a read to the same variable by process P_2, then the two steps or transitions can be swapped. Or generally $\xrightarrow{\alpha}$, $\xrightarrow{\beta}$, $\xrightarrow{\alpha_1}$, etc.

[1] In the context of dynamic logics and multi-modal logics, the transition systems had “multiple” transition relations, which is the same as having labelled transitions.

Apart from that now we consider transition-labels, the definition of transition system or Kripke-structure is basically unchanged.

Definition 6.1.1 (Labelled transition system)**.** A *lablled transition system* T over a set of labels L (also called alphabet) is a tuple $(S, \rightarrow, L, S_0, V)$ where

- S is a set of states.
- $\rightarrow \subseteq S \times L \times S$ is the L-labelled relation between states, the transition relation.
- $S_0 \subseteq S$ is the set of starting states
- $V : S \rightarrow 2^P$ is a map labeling each state with a set of propositional variables.

Determinism and enabledness

In the context of automata (Büchi or otherwise), we have introduced the concept of *determinism.* We use the same definition here for labelled transition systems.

Determinism, generally, applies to situations where the result, the outcome, the nexts state etc. is fixed, as opposed to when more than one result is possible. That would be non-deterministic. Functions are deterministic: the output is determined by the input.

For transition systems (and automata), the conventional definion of determinism is that in a state, the successor state determined, *for a given label.* So it's not as *strict* as the the successor is fixed independent from the action label; that would make the transition system a linear structure.

The action labels, abstractly seen, are elements from some alphabet, but in concrete representations, labels may also be "structured" for instance $W_1(x, 1)$ could represent that process P_1 does some write instruction to variable x, writing value 1, or similar. The label may or may not be seen as input

Definition 6.1.2 (Deterministic)**.** A labelled transition system is *deterministic* if $s_1 \xrightarrow{a} s_1$ and $s \xrightarrow{s_2}$ implies $s_1 = s_2$ (for all states s_0, s_1, and s_2).

See also the depiction in Figure 6.2a later.

In that case (and since we are focusing on deterministic systems): we also write $s' = \alpha(s)$ for $s \xrightarrow{\alpha} s'$ (or $\alpha(s, s')$).

The definition here in the introductory section is not the last word about determinism and related issues in this chapter. Section 6.3.1, there will be variations on the topic.

A subtle point in that context will be the question whether there *is* a next state doing a particular step. If, in a state one can *do* and $\xrightarrow{\alpha}$-transition, one says α is enabled in that state. Otherwise, a is disabled there.

Definition 6.1.3 (Enabled)**.** $\xrightarrow{\alpha}$ *enabled* in s, if $s \xrightarrow{\alpha}$. Otherwise $\xrightarrow{\alpha}$ *disabled* in s.

As a side remark: we will talk about paths π

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

but the paths now are not necessarily infinite. For technical reasons, we focused on infinite paths in LTL (which was not a real restriction, as finite ones could be artificially made infinite by stuttering after reaching the official end).

Concurrency in asynchronous systems

In loosely coupled concurrent systems or asynchronous systems, the actions of different processes running in parallel in an asynchronous fashion are largely independent. In an interleaving model, a concrete run or execution (or path) is an arbitrary linearization. In a single-processor setting it's the scheduler's task to pick at each point one possible next step, i.e., make a choice between the different processes who's next. In a practical schedular, of course it will not make a new pick at each clock tick, but let one picked process run its course until its time slice is up one something else triggers the schedular to give another processs its turn. But in the most extreme case, the scheduler may in principle reconsider the options after each tick. That's often assumed in model checking: one is given the system, but one does not know the scheduler, resp. one wants to verify the system for any possible scheduler (and in isolation, i.e., ignoring other unrelated processes running under the same scheduler.

In a situation where the processes are completelety independent, that leads to a behavior sketched in Figure 6.1 (for actions of three processes, each doing just one step). The actions themselves are assumed atomic.

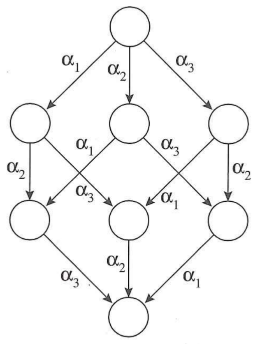


Figure 6.1: Interleaving of three actions / steps

If we have n transition relations (corresponding to n different independent processes, that gives rise to $n!$ different orderings and 2^n states.

Of course that's a very simple case, complete intependence of the relations (and processes). In more realistic situations, some actions or transitions are independent, and some not.

We talked about independence here informally, with an intutive understanding of what it means that processes run indepdently, i.e., have nothing to do with each other. Later we present a definition of *independence* abstractly in terms of labelled transition relations, and not necessarily for actions of parallel processes.

Still, the picture of parallel processes doing steps is useful and gives intuition behind the definitions later.

There will be two aspects of when we call processes (resp. their steps) independent, resp. when they are not independent.

One is that the *order* of steps does not matter, for instance, if two processes write to the same variable at the same time, the outcome will be different depending who comes first. In this situation, both actions or steps can swap their place, i.e., a scheduler can execute them in either order, but the outcome is different. Then the systems would not be considered independent.

But is another aspect: it's still a situation of "who is first" or "whom the scheduler chooses first". However, there is no "second step". So at one point two processes can do a next step, but the one not chosen has lost it's chance. The step that had been *enabled* before the competing situation has been resolved becomes *disabled* by making the decision (that's why we introduced the notion of enabledness for labelled transitions already). At any rate, if the action of one process disables the possible next action of another process, also that breaks independence of course. In terms of concurrent processes, the latter type of non-independence is typical for *synchronization* actions (think lock-taking).

We come back to it in Section 6.3.1

6.2 Pruning the state space: ample sets

Partial order reduction is about reducing the state-space. This section does not provide a concrete solution (for a combination of a concrete programming language and concrete temporal logics). It sketches an skeleton of an idea how achieve the state space reductions.

For the setting, however, we assume a explicit state approach, based on *depths-first search* (DFS).

A super-unrealstic approach would be the following

1. generate explicitly the state space (by DFS for example),
2. then prune it and remove equivalent transitions & states, and finally
3. model-check the property on the smaller space.

Of course it makes no sense to first generate the all of the state space, the trick must be to *avoid* doing that. So a slightly more realistic approach is

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

That's still not realistic; a more practical way will be to both stages at the same time, but for presentational reasons, we focus on generating resp. exploring a reduced state-space. That will be done by modifying a standard depth-first exploration procedure.

Actually, this section here will not provide much more than setting the stage. Let's first recall standard DFS, i.e., the problem to explore or traverse a given graph via a tryically recursive procedure.

We should perhaps be careful when talking about traversing a "given" graph or transition sytem. We do *not* want to imply that the transitition system is stored (via some standard graph representation) in the memory ready to be explored. That would imply the super-unrealistic setting where the whole state space is generated up-front.

Instead, the state-space is explored which being generated (hopefully only partially). In model-checking terms and independent from partial-order reduction, that is known as *on-the-fly* model-checking.

Depth-first search does some recursive state exploration: exploring one given means **visiting all its neighbors or successors**, and doing the same for each of them, remembering which states one has seen already, so that one can stop and backtrack, if one sees a state for the second time, otherwise woul would run into an infitely exploration loop.

One realization of depth-first seach in pseudo-code is shown in Listing 6.1. It's not based on recursive calls of a procedure, but making use of a explicit stack. Furthermore it sketches some way to maintain the states in in some hashed form and organizes the search making use of a work-list. For instance, the basic exploration steps adds successor-steps to the work-list and remove the current action.

Those organasational things are not central for depth-first exploration. The code from Listing 6.1, while still a depth-first strategy however, deviates **crucially** from plain depth-first search. But instead of exploring posible next steps (i.e., all neighbors), the approach realizes the following improvement:

Don't explore *all* enabled transitions, follow **enough enabled** transitions.

That's done in line 6, adding a set of actions to the work-list called **ample.** The word *"ample"* means, especially in this context, something like "enough" or "sufficiently many". In general, **ample** set of transitions in a state $\subseteq$ set of enabled transitions in a state

```
1  hash($s_0$);
2  set on_stack($s_0$);
3  expand_state($s_0$);
4
5  procedure expand_state($s$);
6    work_set($s$) := ample($s$);
7    while work_set($s$) $\neq \emptyset$
8    do
9       let $\alpha \in$ work_set($s$);
10      work_set($s$) := work_set($s$) \{$\alpha$}
11      $s'$ := $\alpha(s)$
12      if   is_new($s'$)
13      then hash($s'$)
14           set on_stack($s'$);
15           expand_state($s'$);
```

```
      end if
    end while
    set completed(s)
end procedure;
```

Listing 6.1: Modified DFS (ample sets)

Requirements on ample sets Of course, on that level, it's *not* a solution to any problem. We have so far not done more than expressing the wish to achieve our goal of pruning the explored state space by a small modification the standard DFS algorithm (and we use the terminology of ample sets to talk about that modification).

The only thing we know is that for each state, the state's ample set of actions is, by definition, a subset of the actions enabled in that state. So the question is: how to restrict the set of enabled transitions (with this restriction called "ample"). There are some *general* requirements on the restriction:

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

The first is a condition sine-qua-non: correctness must not be compromised. How well the remaining to criteria are fulfilled decides on how much the pruning improves the performance. The last 2, however, are somehow in conflict with each other. Investing too much effort in calculating the smallest possible (and still correct) subset may be counter-productive.

The details of a concrete correct and efficient realization of the ample-set idea also depend on the programming or modelling language.

The general idea of ample sets has been explored in the literature in many variations. The names the papers used for their respective proposal are mostly not very indicative of how a particular solution works; names of concepts in well-known approaches include "sleep sets", "persistent sets", "stubborn sets" . . .

6.3 Equivalent behavior

Before looking an algorithmic approach, we loop a bit deeper into *indepence* of systems. We touched upon that already in the introductory remarks of Section 6.1, namely when we consider (the actions of) two or more processes as independen. Here we define more formally the notion of independence of labelled transitions or relations. That can be somehow related also to the question of systems being deterministic, at least deterministic as far as the result is concerned in that the outcome may not depend on the schedule. Therefore we discuss also that. Especially the definition of independence is important for partial-order reduction.

Later a further aspect needs to be taken into account and that has to do with the fact that we are doing verification of a specification (here an LTL formula). Even if the order of actions don't matter with respect to the outcome, still we have to be careful if the order matters wrt. formula, i.e. whether the answer to the model checking question depends on the order. That will be captured by the notion of visibility resp. invisibility.

6.3.1 Equivalent reordering of behavior: Independence

Determinism, confluence, and commuting diamond property

Partial-order reduction works best for asynchronous, loosely coupled systems, as we said, when different parts of the system run independently and without interfering with each other. Of course, the situation where processes run completely and always indepdent are seldom. Resp. they are uninteresting. If parts of the system are truly independent, there is no need to jointly model them and jointly verify them.

In some way, such a setting with, say two completely independent processes, would be ideal for partial-order reduction. A really optimal reduction could reduce the system to doing the steps of one process first, and then doing the other afterward. If each process is in itself deterministic, the reduced system doing first P_1 and then P_2, would show a linear, deterministic behavor, and no combinatorial state explosion due to unterleaving the steps of the processes when exploring $P_1 \parallel P_2$ ($\parallel$ here for the asynchronous parallel composition).

But of course, it's a stupid idea to try to jointly analyse $P_1 \parallel P_2$ and hope some mystical POR-model-checker will comes up with an ideal reduction (which is unrealistic) if one already knows that P_1 and P_2 are independent.

More interesting are processes that interact from time to time, working on shared variables, exchanging messages etc, but also work independently for large stretches of time. Doing independent actions means that the outcome does not depend on the order of the actions.

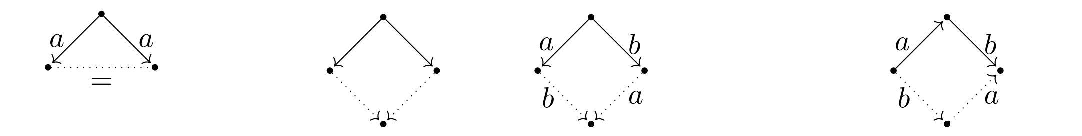
Assume that processes are internally deterministic, i.e., there are no internal sources of non-determinism. That means, the "outcome" can only be influenced by the way the actions of different processes are interleaved. In case of independent, actions, the outcome remains the same, as there order can be inverted. In the (unintesting) case of completely independent processes, also the outcome is the same, whether we do $P_1; P_2$ or $P_2; P_1$: every (deterministic) process calculates its own result, it's only a question who comes first.

This is to say the notions of determinism and independence are closely connected. They are not the same though (and we have not formally defined what independence actually means). For a transition system to be deterministic, remember see Definition 6.1.2.

The version of POR and the notion of ample sets will be based on a so-called *independence relation* (see Definition 6.3.1). Like ample-sets it will be rather non-concrete. I.e., it will not specify what concrete actions (reading, writing, synchronizations etc.) in some setting *are* independent. It spells out more general conditions as to when a relationship between actions or transitions qualifies to be called an independence relation.

Before doing that, let's at least have a short look at aspects relating "independence" of actions and determinism, because the notions are in spirit similar, but technically not the same.

We will discuss it mostly abstractly, i.e., as a property of relations or pairs of relations. Assume we are dealing with a labelled transition system.



(a) Determinism (b) Diamond property (c) comm. d-property (d) swap./commuting

Figure 6.2: Different flavors of determinism and "order does not matter"

and vice versa. We assume that the transition relations $\xrightarrow{\alpha_i}$ are *deterministic*, and we write $\alpha_i(s)$ for $s \xrightarrow{\alpha_i}$.

Definition 6.3.1 (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 $\alpha_1, \alpha_2 \in \mathit{enabled}(s)$, then $\alpha_1 \in \mathit{enabled}(\alpha_2(s))$

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

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

The complement of a independence relation is called, not surprisingly, a *dependence relation*. I.e., given some independence relation I then $D = (\rightarrow \times \rightarrow) \setminus I$ is a dependence relation.

Is that all?

Let's look at the commuting diamond diagram again, from Figure 6.2c, resp. repeated in Figure 6.3. We made arguments that situations of that form have the potential for saving since the order of the two steps does not matter, since each order, they reach the same state, r in Figure 6.3.

There are, however, two issues or complications to take into account

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.

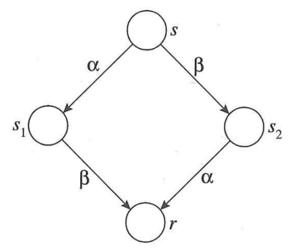


Figure 6.3: Commuting diamond

6.3.2 Equivalence wrt. formula(s): visibility and stuttering

The first point is an aspect we have ignored in the discussion so far, namely the influence of the property or the class of properties to check. However, in a situation like the communting diamond, of one wants to explore the path via s_1 only, ignoring the one over s_2, it must be sure that satisfaction of the property to check does not depend on visiting those states. For instance, if there is a proposition that holds in s_1 but not in s_2 or vice versa, then one has to explore both alternatives. If propositions can make differentiate between two states, one says, the formula can "see" the difference or that the two states resp. their difference is *observable* or *visible* by the formula. The picture behind that terminology is that checking for a formula is a way to observe a system. Behavior that cannot make a difference whether the formula holds or not is not observable or invisible. That terminology is often used for classes of formulas or whole logics. More expressive logics can observe more behavior, resp. differentiate between more systems that weaker ones. Weaker ones, in that sense, have more potential for reductions, here partial-order reduction.

Here, for our purposes, we define when to call a *transition* as being (in-)visible for a set of propositional atoms.

Definition 6.3.2 (Propositional vsibility)**.** Assume a valuation $V : S \to 2^P$. The transition relation $\xrightarrow{\alpha}$ is *invisible* wrt. a set of $P' \subseteq P$ if for all $s_1 \xrightarrow{\alpha} s_2$ and all $p \in P'$, the following holds

$$s_1 \models p \quad \text{iff} \quad s_2 \models p \tag{6.1}$$

The definition means, that no α-transition changes the truth-status of an of the propositional variables in P'.

So, state changes which don't change the truth status of any propositional variable (in P') is invisible (wrt. P'). It's a step that does not matter concerning the question whether the transition system satisfies the specification or not. Steps that don't matter are also called *stutter steps* and the phenomenon *stuttering*.

We have encountered stuttering already in connection with LTL and the definition of π. Paths, in that context, were defined as being necessarily infinite. To accomodate terminating behavior of a system, we extended an terminating behavior by an infinite sequence of stuttering steps at the end, to obtain an infinite path.

Of course, it is possible that the system does some "actual" stuttering while still running, doing stretches of invible steps. Those stretches of stuttering steps, resp. maximal such stretches, are sometimes called *blocks*. Note in passing: terminating behavior, which stuttering at the end, consists of a finite number of blocks, since the stuttering extension after termination is represented by one infinitely long block.

Two paths that consist of the "same" sequence of blocks are called stuttering-equivalent. When saying the "same" sequence of block, we mean that each block of π is represented by a corresponding block in π' with the same propositional valuation, but the size of the block, i.e., the number of stuttering steps inside the block may different. This is shown in Figure 6.4. We write $\sim_{st}$ for stutter-equivalence for paths.

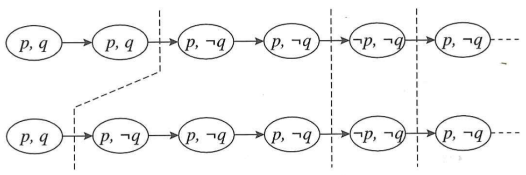


Figure 6.4: Stuttering-equivalent path

The different states in a block have the same status as far as the satisfaction of propositional variables is concerned. That carries over to general propositional formulas, of course. But what about temporal properties of LTL? Does an LTL formula that holds for some path also hold for a stutter-equivalent one, does $\pi \models \varphi$ and $\pi \sim_{st} \pi'$ imply $\pi' \models \varphi$. To say it differently: can the addition or removal of stutter steps in a π change the truth status of the path with respect to a temporal property.

Before we closer at that, let's first give a name to LTL formulas whose truth-status is untouched by adding or removing stuttering steps.

Definition 6.3.3 (Stutter invariance)**.** An LTL formula φ is *invariant under stuttering* iff for all pairs of paths π_1 and π_2 with $\pi_1 \sim_{st} \pi_2$,

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

Using the observability-terminology, one could say that for those formulas stuttering-steps are invisible. Technically, we have introduced the notion of invisibility for propositional proporties only (and we defined it for transitions $\xrightarrow{\alpha}$ in a transition system) in Definition 6.3.2, but conceptually it's analogous.

Back to the question: can additional stutter steps in a π change the truth status of the path with respect to a temporal property, i.e., are LTL formulas stutter invariant (or at least some). We clarified already that for propositions are stutter invariant, but that's not very interesting and helpful.

However, adding (or removing) a stutter steps may change satisfaction of $\bigcirc$-*formulas!*. That's because blocks are *finite*. Assume that some $\bigcirc p$ us be true at *last* position of a block and whether it's true or not depends on the situation in the subsequent block.

Therefore, extending the block longer by adding a stutter step may change the truth status in the considered position. Similar for removing stutter steps.

It turns out, if we banish $\bigcirc$ from LTL, formulas become stutter invariant. The truth status of $\Diamond$ and $\Box$ (and for the more complex binary operators) does not depend on the length of the stutter blocks. For example $\Diamond$ refers to some finite point in the future, and adding or removing stutter steps does not change that. Note that stuttering means adding or removing only *finitely* many steps. That means, stuttering cannot turn a finite block into an infinite one.

The restricted form of LTL is known as next-free LTL, and abbreviated as LTL$_{-\bigcirc}$(or also LTL$_{\text{-X}}$).

LTL$_{-\bigcirc}$ is stuttering invariant. That's good for reducing the state space, because it gives hope to ignore states, ideally to take only one representative in each block. And the more stuttering and the larger the blocks, the more potential for space reduction.

We defined the concepts like visibility and stutteing on transition systems and paths. Those stem of course from the description of a concurrent system, consisting of processes or threads running in parallel. As a general observation: the more loosely coupled or more asychronous a system is, the more one can expect to see stuttering. That's probably not self-evident, it's also not a mathematical fact. It's also not just a consequence on the loose coupling of the system, but also based on how one typically specifies properties for such loosely coupled system.

Of course the logic because less expressive that way. That's in a way the point, making it less expressive makes stutter steps unobservable.

stuttering (in this form): important for *asynchronous* systems . . .

6.4 POR for LTL$_{-\bigcirc}$

LTL$_{-\bigcirc}$ is is one useful and fuitful general setting for POR (especially for asynchronous systems), so let's look at that. Partial-order reduction and related ideas have also been studied from different angels and settings and logics. One may also try to achieve reductions for, say, safety-properties, only,, or even try with reductions per formula. But we do it for LTL$_{-\bigcirc}$, and we want to do that in terms of the *ample-set* variant of depth-first search.

Given a transition system T, let's refer to the reduced or pruned version as $T^{\text{✂}}$, reduced in the sense containing only the part of the state space explored via the neighbors in the ample-sets.

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

Since we a doing a form of LTL, the correctness is mainly a condition on *paths,* i.e., all the path in T starting from s. So the correctness is assured if each path in the original system is equivalently represented after pruning, at least once.

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

6.4.1 Conditions on selecting ample sets

The correctness condition spell out for path is straightforward and should be obvious. Far less obvious is how to achieve that by doing a proper definition of the ample set. Of course, if correctness were the only goal, that could be trivially achieved by setting the ample set to be the equal to the set of enabled transition in each state. Then there would be no reduction whatsoever, and that is of course correct. The trick must be to make the ample set as small as (reasoably) possible, without compromising correctness.

In the design of the algorithm, it's about making a selection of enabled steps and following those, leaving out others., local decisions, paths . . .

reorderings of

- each pruned path can be "reordered" to an which is explored (using *independence*). It also includes a condition covering end-states. See Section 6.3.1.
- make sure that the reordering (pre-poning) does not change the logical status, based on the notions of *stuttering* and *visibility*. See Section 6.3.2.
- "fairness": make use not to prune "relevant" transitions by letting the search **cycle** in irrelevant ones.

We will later show code snippets covering those conditions.

Reordering conditions ($\mathbf{C}_0$, $\mathbf{C}_1$)

This section is making sure that, when cutting out a path or rather a whole set of paths by not exploring some neighbors, the omitted paths are covered equivalently otherwise.

This requirement is split into to conditions. The first one, let's call it $\mathbf{C}_0$ is very trivial. We know that $ample(s) \subseteq enableds$ and thus if $enabled(s) = \emptyset$, also the ample set is of course empty: if there are no neighbors, one can not continue exploring anyway and the depth-first seach backtracks. Note that it's not about that there are no more *unexplored* neighbors; also in this case the exploration backtracks. It's about having reached a dead end.

But if there are enabled steps, i.e. $enabled(s) \neq \emptyset$, then it's clear that we cannot set the ample-set to $\emptyset$ as well. Without at least following one of the possible neighbors, there's no way to know what would have happened if we followed at least one. So we have to require the following:

C$_0$: stop at dead ends, only.

$$\mathit{ample}(s) = \emptyset \text{ iff } \mathit{enabled}(s) = \emptyset \tag{6.2}$$

The next condition is a bit more tricky. It's also not formulated in an actionable or readily implementable form. It spell out just a condition on the ample sets in a state connection with paths that start in that state.

C$_1$ Along every path in T starting at s, the following condition holds: a transition **dependent** on a transition in $\mathit{ample}(s)$ cannot be executed without a transition from $\mathit{ample}(s)$ occuring *first*.

Observation 6.4.1. Under condition **C**$_1$, we have:

$$\mathit{ample}(s) \bowtie \neg\mathit{ample}(s)\ . \tag{6.3}$$

As a consequence of the definition: in a state, all enabled ones but not ample are independent from the ample ones. 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.

Another way of saying the same is

> The set $\mathit{ample}(s)$ is closed under #

As a consequence of **C**$_1$, one can distinnguish two forms of paths, with respect of which actions from within and outside the ample-set they contain. These two forms are

$$\beta_0\beta_1\ldots\beta_m\alpha \quad \text{or} \quad \beta_0\beta_1\beta_2\ldots \quad \text{with} \quad \alpha \in \mathit{ample}(s) \quad \text{and} \quad \beta_i \bowtie \mathit{ample}(s)\ . \tag{6.4}$$

In the first case, the path has a finite prefix $\beta_0\beta_1\ldots\beta_m\alpha$, i.e., after a few β_i independent from all actions in the ample set, at some point a member of the ample set (of s) is taken. In the second case, there is an infinite sequence $\beta_0\beta_1\beta_2\ldots$ where also $\beta_i \bowtie \mathit{ample}(s)$.

Note that the two case are *not* mutually exclusive. Remember that the ample sets are closed under #, thanks to **C**$_1$, but the condition is a loose one. It states that if a action is outside in the ample set, it has to be independent from it. However, it does not forbid to put independent actions inside.

However, without loss of generality, we can focus on the case that all $\beta_i \notin \mathit{ample}(s)$, in which case the two conditions from equation (6.4) are mutually exclusive. Either there is eventually an α from the ample set of s preceded by β's outside that ample set. Or there are infinitely many such β's. As a side remark, in that form, it is a form of weak-until property of the paths.

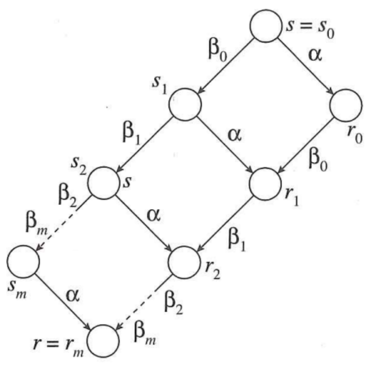


Figure 6.5: $\mathbf{C}_1$ (reorder) and $\mathbf{C}_2$ (invisibility)

We can now make an argument that we can reorder paths appropriately, using *commutation.* Consider the situation in Figure 6.5, which corresponds to the first case of equation (6.4). The assumptions are, as agreed, that $\alpha \in \mathit{ample}(s)$, that $\beta_i \notin \mathit{ample}(s)$,

Consider further the two paths starting in s,

$$\pi_1 = \vec{\beta}\alpha \quad \text{and} \quad \pi_2 = \alpha\vec{\beta} \ .$$

As far as the unreduced transition system is concerned, we have that $\pi_1 \in T, s$ implies $\pi_2 \in T, s$ (and vice versa). That's a direct consequence from the fact that α and the β's are independent (see Definition 6.3.1).

Concerning $T^{✂}$, we have under the given assumptions,

$$\pi_1 \notin T^{✂} \quad \text{and} \quad \pi_2 \in T^{✂} \ ,$$

at least if we assume $m > 0$, excluding the uninteresting and trivial corner case that there are no β's at all. So it's an example of an actual reduction, the pruned system leaves out π_2 but explores the reordered one

Invisibility ($\mathbf{C}_2$)

So, as far as their *existance* in T is concerned, π_1 and π_2 are "equivalent" (and all the "intermediate" paths as well, like $\beta'\alpha\beta''$). 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. the intermediate, visisited states, 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 state r_i or the state 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)

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.

C$_2$ (invisibility):

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

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.

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

C$_3$ (cycle condition)

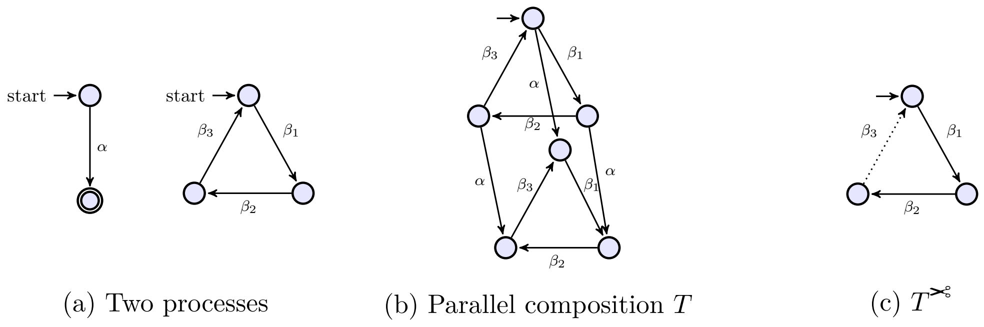
The previous condition **C**$_2$ insisted on invisiblity of an action α, in case one omits alternatives. The transition system from Figure 6.5 shown previously illustrated that, that if α is invisble, the uncovered path (in the picture) with α at the end can be reordered with α at the beginning *without* omitting intermediate states with different logical status. That last condition about invisibility took care about *one* form of paths from equation (6.4) that follow as consequence of condition **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 **C**$_3$ is about the second form of paths from equation (6.4), namely the ones with infinitely many β_i, and *never* any α from the ample set.

Based on 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 $\neg\mathit{ample}(s)$ are the one that are 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 “this transition needs to be explored”. Postponing it forever is not the way to go.

Like the other conditions as well, condition $\mathbf{C}_3$, is not a condition on the behavor or the form of paths (like "don't look at paths where transitions $\alpha \in \mathit{ample}(s)$ are postponed forever"), it's a 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).

Let's look at Figure 6.6 The three figures serve to illustrate the previously discussed problem of "infinite postponement". To complete the example, we need to add one piece of information, namely the "logical part", i.e., at which states satisfy which propositions.



(a) Two processes (b) Parallel composition T (c) $T^{\text{✂}}$

Figure 6.6: Illustration of the cycle condition

Figure 6.6a contains two separate processes and Figure 6.6b their (asynchronous) parallel composition. In the example, assume that α is *visible*, the β_is are *invisible*. We also assume that the α is *independent* from all the β's. With all its transitions invisible, the second process is stuttering. That means also, that, as far as the property to be model-checked is concerned, the focus is on the first process, the second one is irrelevant as far as the property is concerned.

Since the actions of the two processes are also indepdent, the two processes are completely independent. There is no synchronization between them (independence) and they are not "secretly" couple via the property (invisibility). The example is this pretty trivial, but it's used to illustrated the need for the last condition.

More concretely, the picture may repesent a situation, where there is one boolean variable, initially say "false", and the process to the left sets it to "true" via it's transition $\xrightarrow{\alpha}$, i.e., the label α may represent the assignment `p := true`. The other process does not do anything (except spinning around, cycling through its three states), resp. does nothing interesting as far as the property to be checked is concerned. For instance, the property could be $\Diamond(p = \mathit{true})$, and the second process only operates on variables other than p

Figure 6.6b shows the two processes running in parallel in an asynchronous fashion, i.e., interleaving their steps. The overall combined behavior is given by the transition system T, with 6 states. In that T 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 T, not in $T^{\text{✂}}$). 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 $\mathbf{C}_2$.

In contrast to the β's, transition α *is* visible, it does not stutter, so taking it matters wrt. the verification problem. However, the ample sets chosen as given, leads to explorations in $T^{\text{✂}}$ *ignoring* α.

The last condition $\mathbf{C}_3$ excludes 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, infinite paths must come from running through at least one cycle. What needs to be ensured is that a situation as in the example cannot occur. That $\xrightarrow{\alpha}$ 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. If left out completely 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 some edge itself is invisible, 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 $\xrightarrow{\alpha}$ that 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.

Side remark 6.4.2 (Zeno)**.** A final side remark (not too 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. □

Now, without furter ado, here's the condition

$\mathbf{C}_3$ (cycle condition):

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

With all the conditions nailed now, let's go back to the two issues we sketched in connection with the commuting diamond Figure 6.3. As a recap: the two issues mentioned where:

1. Does the satisfaction depends on chosing the path via s_1 or via s_2?
2. When following only one path, do we forget to check successors?

Let's focus on the second issue and let's look at Figure 6.7, where s_1 had successor(s) reachable via transition γ. Assume the state s_1 is omitted, i.e., $\beta \in \mathit{ample}(s)$, but not α.

So, by omitting s_1, do we forget to check parts of the system?

the conditions imply

$$\begin{array}{rcl} ss_2r & \sim_{st} & ss_1r \\ ss_1s_1' & \sim_{st} & ss_2rr' \end{array}$$

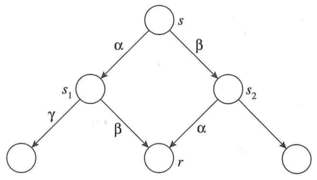


Figure 6.7: Forgotten successors?

Calculating the ample sets

We don't go much into details here, but looking at the different conditions, it's clear that they quite different in complexity. The conditions need to be checked *on-the-flow* during the depth-first exploration. Therefore checking the condition can prefably be done efficiently.

$\mathbf{C}_0$ and $\mathbf{C}_2$ are easy. More tricky is $\mathbf{C}_1$. Note that the condition refers to T, to to $T^{✂}$. It is equivalent to reachability checking. As for $\mathbf{C}_3$, also that is tricky. One can replace the condition by a modified one, that is easier to establish namely

> 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*

Applying the ample-set theory of POR

The whole presentation of partial-order reduction based on ample-sets is rather abstraction. It speaks about multiple relations, i.e., the labelled transitions and conditions on them, like being independent resp. when they are inter-dependent.

In practice actions correspond to atomic steps in a (concurrent) program or a model thereof. It's also important that those steps are also *deterministic*, since one general assumption underlying the whole setting is that the labelled transition system is deterministic. If that's not the case, the framework does not work in the presented form, resp. need to be refined.

That's typically not big restriction, atomic individualy steps are deterministic. A source of non-determinism, however, could be is abstraction, for instance in that the model abstracts away from details of a concrete programs. Also that actually is not really problematic for the POR framework.

Anyway, we

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

Let's fix some notations and definitions. We write now α for $\xrightarrow{\alpha}$. We assume a fixed, finite set of processes P_i or just i for short. We also write T_i: those transitions that "belong to" P_i

In the abstract discussions we referred to states of a transition system by s or simulir. Now, the states are assumed to have more internal structure. Since the program is seen as the parallel consposition of a number of processes. The overall state contains then the tuple of the states of the individual processes. Each process is a particular location, or *control-flow point* in its own execution. If the control-flow is depicted as some transition system (or control-flow graph), that control-flow point corresponds to one node in that transition system. We can also see it as the current value of the *program counter*. For notation, given a global

Definition 6.4.3 (Referring to parts of a global state)**.** Let $pc_i(s)$ be the value of the program counter of process i in state s. We refer by $T_i(s)$ to the set of enabled transitions in state s enable in process P_i

Definition 6.4.4 (Relationships between actions)**.** $dep(\alpha)$: transitions interdependent with α $pre(\alpha)$: transitions whose execution *may* enable α

can be over-approximative

When are transitions (inter)dependent The definition of independence has two aspects, commutativity is one. The other one concerns enabledness. Basically, it's about *preserving* enabledness, resp. to forbid an action to disable the other. Since the definition of independence is symmetric, this condition works both ways. For α_1 and α_2 being independent means that for all states where both are enabled, taking α_1 must not disable α_2, and vice versa.

Being inter-dependent means that communtativity or the enableness condition breaks, at least in one state. For the condition that independence breaks in a least one state, it's however likely that if two enabled actions don't commute on one state, the don't commute in a different states where they happen to be also enabled. Same for breaking the preseving-enabledness condition. After all, actions like reading from or writing to memory or other forms of interactions, work uniformely.

We said that $\alpha_1 \# \alpha_2$ mean violating commutativity or disabling the competitor. Actually, it's unlikely or unrealistic to find examples of actions which violate both. The reason is simple: assume that taking α_1 disables α_2. Then j$\xrightarrow{\alpha_1}\xrightarrow{\alpha_2}$ does not occur, with α_2 disabled after taking α_1. That means, $\alpha_2(\alpha_1(s))$ does not exists or is undefined. Then one could make the argument, that breaks the second requirement for indepedence

$$\alpha_2(\alpha_1(s)) = \alpha_2(\alpha_2(s)) \tag{6.5}$$

becaise the left-hand side is undefined and the right-hand side may still be ok. To break independence, it's enough that α_1 disables α_2, it's not needed that α_2 also does the same with α_1. In practice, when an action disables another, the situation is symmtetric. It describes a *choice point* in a program where one of two (or more) actions is taken and the others disabled. Choice-point not in the sense of a case-construct of a sequential program. Typically would be *lock-taking* actions. The hole purpose of a lock is to do exactly that (to ensure mutex): let at most one action succeed, and disabling competitors, at least for some time, until the lock becomes free again.

But the behavior of the lock-taking actins is symmetric, if process P_1 takes the lock at a place where P_2 could also take it, P_2 is disabled, and vice versa. That means, both sides of equation 6.5 don't exists or are undefined.

Assuming that in practice such disabling actions are symmetric, there could be another reason why in a situaton $\alpha_1 \# \alpha_2$, both swapping and the enabledness condition are violated. That has to do with the fact that one needs to find only one state where the conditions for $\alpha_1 \bowtie \alpha_2$ don't hold. And it could be that in one state, commutativity is violated, and in another one the enabledness part. So that would be another situation when both fail, namely at different states. But again, that unplausible in practice. Actions typically act uniform in states.

Transitions that may enable α ($pre(\alpha)$) Assume α is an action from one specific process P_i

$$pre(\alpha) \supseteq \{\beta \mid \alpha \notin enabled(s), \beta \in enabled(s), \alpha \in enabled(\beta(s))\} \tag{6.6}$$

- $pre(\alpha)$ 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

6.4.2 Some code snippets for the conditions

We have discussed the different conditions in some detail and mentioned a bit how they can be checked. Here, for completeness, some pseudo-code that sketch how to algorithmically determine the conditions. For the third condition, the code shows the variant $\mathbf{C}_3'$ mentioned shortly earlier which is easier to check than the more precise one $\mathbf{C}_3$.

```
function ample ($s$) =
 for all $P_i$ such that $T_i(s) \neq \emptyset$ // try to focus on one $P_i$
   if
         check_C1($s, P_1$) $\wedge$
         check_C2($T_i(s)$) $\wedge$
         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
```

Listing 6.2: ample

```
function check_C2($X$) =
  for all $\alpha \in X$
  do  if   visible($\alpha$)
      then false
      else true
```

Listing 6.3: Check $\mathbf{C}_2$

```
function check_C3' ($s, X$) =
  for all $\alpha \in X$
   do
      if   on_stack($\alpha(s)$)
      then false
      else true
```

Listing 6.4: Check $\mathbf{C}'_3$

```
function check_C1 ($s, P_i$) =
  for all $P_j \neq P_i$
   do
      if          $dep(T_i(s)) \cap T_j \neq \emptyset$
          $\vee$
                  $pre(current_i(s) \setminus T_i(s)) \cap T_j \neq \emptyset$
      then return false
   end forall;
  return true
```

Listing 6.5: Check $\mathbf{C}_1$

Bibliography

[1] Baier, C. and Katoen, J.-P. (2008). *Principles of Model Checking*. MIT Press.

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

[3] Diekert, V. and Muscholl, A. (2011). Trace theory. Available on the internet.

[4] Peled, D. (1994). Combining partial-order reduction with on-the-fly model-checking. In Dill, D., editor, *Proceedings of CAV '94*, volume 818 of *Lecture Notes in Computer Science*, pages 377–390. Springer Verlag.

[5] 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.

[6] 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