Course Script

IN 5110: Specification and Verification of Parallel Systems

IN5110, autumn 2021

Martin Steffen

Contents

Chapter 3
LTL model checking

What is it about?

Learning Targets of this Chapter

The chapter covers LTL and how to do model checking for that logic, using Büchi-automata.

Contents

3.1 Introduction

In this chapter, we leave behind a bit the classical "logical" treatment of logics like asking for validity etc., i.e., asking $\models \varphi$, but proceed to the question of *model checking*, i.e., when does a concrete model satisfies a formula $M \models \varphi$ (more precisely, when does a state s in a model satisfies a formula, written $M, s \models \varphi$). We do that for a specific modal logic, indeed, a specific temporal logic. It's one of the most prominent ones and the first one that was taken up seriously in computer science (as opposed to studied in logics, mathematics or philosophy). We will also cover a central way of doing model checking of such temporal logics, namely *automata-based* model checking.

3.1.1 Temporal logic?

Temporal logic is a modal logic of "time". There is not just one temporal logic, there are in fact many. Time in that context is mostly not understood as the real-world time, like time of the day in hours and seconds, though there are temporal logics to deal with that.

Time is understood more abstractly, capturing more *changes* in a system or situation but abstracting away from when exactly that happens. Even if one abstracts away from that, there aare different ways of modelling time. One inportant distinction is that of **linear** vs. **branching** time.

The linear picture is probably more how one informally thinks about time. Each day is followed by the next one, time flows from the past to the future, that's the linear picture behind a *time line*. Systems, in particular concurrent systems, are *non-deterministic*: running it multiple times will result in different outcomes. That can come fron internal reasons, like different scheduling decisions or other sources of internal non-determinism, But it can also be cause by interacting with the outside world. That's typical for reactive or interactive systems and different interactions leads to different reactions as well (external non-determinism). Whatever reasons for the non-determinism, if one repeats running a system, the run or execution, seen as a linear sequence of steps, will be different. And typically, there are very many different runs.

But still, that's a *linear* picture. The behavior of a system is characterised by a set of runs or executions (typically a huge number, maybe infinite). *Branching* time takes a more complex view. In that view, the non-determinism of the system is not just captured by the fact that there are many different linear runs, but by modelling that a each point in time (or a least some intermediate points in time) the system can behave non-deterministically in that it could continue in different ways: the continuating branches. Repeating that pattern leads to a tree-like model, which underlies branching time. We see both models and logics for both models.

Besides that, there are other aspects how to treat time. For instance, *discrete* time vs. *continuous* time. We mostly deal with discrete time. Systems proceed step by step, from one state to other, and that's discrete behavior. Time can be see as time *instances* or points in time (as we mostly do), but there are also logics and models what work with time *intervals*.

Another distinguishing characteristic is the following. Most logics will be able to express properties concerning the *future*. Like: "never will there be a deadlock", meaning "never in the future". But there also also versions that can (additionally) talk about the past.

The notion of *time* here, in the context of temporal logics in general and LTL in particular, is kind of abstract. Time is handled in a similar way as we did when introducing modal logics in general, i.e., as "relation" between states (or worlds): proceeding from one state to another via a transition means a "temporal step" insofar that the successor state is "after" the first state. But the time is not really measured, i.e., there is no notion of how long it takes to do a steps. So, the systems and correspondingly the logics talking about their behavior are not *real-time* systems or real-time temporal logics. There exists, however, variants of temporal logics which handle real-time, including versions of real-time LTL, but they won't (probably) occur in this lecture.

3.2 Linear-time temporal logics

In **linear temporal logic (LTL)**, also called *linear-time temporal logic*, is one of the most prominent temporal logics used in model checking. It's supported notably by Spin, and other model-checkers. It was the first, or one of the first temporal logics taken up in computer science, and there are variations of that logics.

With it, we can describe properties like, for instance, the following: assume time is a *sequence* of discrete points i in time, then: if i is *now*,

- p holds in i and every following point (the future)

- p holds in i and every preceding point (the past)

$$\begin{tikzcd}\dots \arrow[r] & \bullet^{p}_{i-2} \arrow[r] & \bullet^{p}_{i-1} \arrow[r] & \bullet^{p}_{i} \arrow[r] & \bullet^{p}_{i+1} \arrow[r] & \bullet^{p}_{i+2} \arrow[r] & \dots\end{tikzcd}$$

Time here is *linear* and *discrete*. One can consequently just use ordinary natural numbers (or integers) to index the points in time. We will mostly be concerned with properties referring to the future, i.e., we won't go much into past-time LTL resp. versions of LTL that allow to speak about the future *and* the past.

3.2.1 Syntax

As before, we start with the syntax of the logic at hand, it's given by a grammar, as usual. We assume some underlying "core" logic. Focusing on the temporal part of the logic, we don't care much about that underlying core. Practically, when it comes to automatically checking, the choice of the underlying logic of course has an impact. But we treat the handling of the underlying logic as *orthogonal*. We will mostly we just assume *propositional* logic as core logic, but the LTL-part of the story would not change if we used first-order logic or another logic.

The first thing to extend is the syntax: we have formulas ψ of said underlying core, and then we extend it but the temporal operators of LTL, adding $\Box$, $\Diamond$, $\bigcirc$, U, R, and W. So the syntax of (a version of) LTL is given by the grammar of Table 3.1.

ψ			propositional/first-order formula
φ	$::=$	ψ	formulas of the "core" logics
	$\mid$	$\neg\varphi \mid \varphi \wedge \varphi \mid \varphi \rightarrow \varphi \mid \dots$	boolean combinations
	$\mid$	$\bigcirc\varphi$	next φ
	$\mid$	$\Box\varphi$	always φ
	$\mid$	$\Diamond\varphi$	eventually φ
	$\mid$	$\varphi\ U\ \varphi$	"until"
	$\mid$	$\varphi\ R\ \varphi$	"release"
	$\mid$	$\varphi\ W\ \varphi$	"waiting for", "weak until"

Table 3.1: LTL syntax

As in earlier logics, one can ponder, whether the syntax is *minimal*, i.e., do we need all the operators, or can some be expressed as syntactic sugar by using others? The answer is: the syntax is *not minimal*, some operators can be left out and we will see that later. For a robust answer to the question of minimality, we need to wait until we have clarified the meaning, i.e., until we have defined the semantics of the operators.

Remark on the syntax We had mentioned it already earlier, in the section covering modal logics. The operators always and eventually from LTL are written often as $\Box$ and $\Diamond$, but they are interpreted slightly different from the interpretation of box and diamond in conventional modal logic. Since we are working in a *linear* structure, the $\bigcirc$-operator corresponds to both box and diamond over the transition relation, both collapsed. One can also interpret $\Box$ and $\Diamond$ from LTL to correspond to the traditional modalities over the **transitive closure** of the one-step successor relation.

3.2.2 Semantics

For the semantics, we need to define a satisfaction relation $\models$ between "models" and LTL formulas. In principle, we know how that works, having seen similar definitions when discussing modal logics in general (using Kripke frames, valuations, and Kripke models).

Now, that we are dealing with a *linear* temporal logic, the Kripke frames are of linear structure. As usual, what kind of *valuations* we employ would depend on the underlying logics. For example for propositional LTL, one needs an interpretation of the propositional atoms per world, for first-order LTL, one needs a choice of the free variables in the terms and formulas (the signature and its interpretation does not change when going from one world to another, only, potentially, the values of the variables).

That's also what we do next, except that we won't use explicitly the terminology of *Kripke frame* or Kripke model. We simply assume a sequence of discrete time points, indexed by natural numbers. So the numbers i, $i+1$, etc. denote the worlds, and the accessibility relation simply connects a "world" i with its successor world $i+1$.

As was done with Kripke models, we then need a valuation per world, i.e., per time point. In the case of propositional LTL, it's a mapping from propositional variables to the boolean values $\mathbb{B}$. To be consistent with common terminology, we call such a function of type $P \to \mathbb{B}$ here not a valuation as we did mostly before, but a **state** (but see also the side remarks about terminology below). Let's use the symbol s to represent such a state or valuation. A *model* then provides a state per world, i.e., a mapping

$$\mathbb{N} \to (P \to \mathbb{B}) \ . \tag{3.1}$$

This is equivalently represented as an infinite sequence of the form

$$s_0 s_1 s_2 \ldots \tag{3.2}$$

where s_0 represents the state at the zero'th position in the infinite sequence, s_1 at the position or world one after that, etc. Such an infinite sequence of states is called **path**, and we use letters π, π' etc. to refer to paths. It's important to remember that paths are *infinite*.

Some remarks on terminology: paths, states, and valuations The notions of states and paths ... are slightly differing in the general literature. It's not a big problem as the used terminology is not incompatible, just sometimes not in complete agreement.

For example, there is a notion of path in connection with graphs. Typically, a path in a graph from a node n_1 to a node n_2 is a sequence of nodes that follows the edges of the given graph and that starts at n_1 and ends in n_2. The length of the path is the number of *edges* (and with this definition, the *empty* paths from n to n contains one node, namely n). There maybe alternative definitions of paths in "graph theory" (like sequences of nodes instead of edges). In connection with our current notion of paths, there are 3 major differences. Our paths are *infinite*, whereas when dealing with graphs, a path normally is understood as a *finite sequence*. There is no fundamental reason for not considering (also) infinite paths in graphs (and some people of course do), it's just that the standard case there is finite sequences, and therefore the word *path* is reserved for those. LTL, on the other hand, deals with ininite sequences, and consequently uses the word paths for those.

The other difference is that a path here is not defined as "a sequence of nodes connected *by edges*". It's simply an infinite sequence of valuations (and the connection is just by the position in the sequence), there is no question of "is there a transition from state at place i to that of at place $i+1$ (or one may see it as implicitly given by the "underlying" implict linear Kripke-frame, where there is an edge from i to $i+1$). Later, when we connect the current notion of paths to "path through a transition system", then the states in that infinite sequence need to arise by connecting transistions or edges in the underlying transition system or graph.

Finally, of course, the conventional notion of path in a graph does not speaks of valuations, it's just a sequence of nodes. If N is the set of nodes of a graph, and $\mathbb{N}_n$ the finite set $\{i \in \mathbb{N} \mid i < n\}$, then a traditional path (of length n) in graphs is a function $\mathbb{N}_n \to N$ such that it "follows the edges".

There are other names as well, when it comes to linear sequences of "statuses" when running a program. Those include *runs*, *executions* (also traces, logs, histories etc.). Sometimes they correspond to sequences of edges (for instance, containing transition labels only). Sometimes they correspond to sequences of "nodes" (containing "status-related" information like here), sometimes both.

Anyway, for us right now and for propositional LTL), a path π, as given in Definition 3.2.1 is an infinite sequence of states (or valuations).

Paths and computations

Definition 3.2.1 (Path)**.** A *path* is an infinite sequence

$$\pi = s_0, s_1, s_2, \ldots$$

of states. It can be seen as a mapping of type $\mathbb{N} \to (P \to \mathbb{B})$.

π^k denotes the suffix path $s_k, s_{k+1}, s_{k+2}, \ldots$. π_k denotes the state s_k.

It's intended that (later) paths represent behavior of programs resp. "going" through a transition system. A transitions system is a graph-like structure (and may contain cycles), and a path can be generated following the graph structure. In that sense it corresponds to the notion of *paths* as known from graphs (remember that the mathematical notions of graph corresponds to Kripke frames). Note, however, that we have defined path *independent* from an underlying program or transition system. It's not a "path through a transition system", but it's simply an infinite sequence of state (maybe caused by a transition system or maybe also not).

Now, what's a *state* then? It depends on what kind of LTL we are doing, basically propositional LTL or first-order LTL. A state basically is the interpretation of the underlying logic in the given "world", i.e., the given point in time (where time is the index inside the linear path). In propositional logic, the state is the interpretation of the propositional symbols (or the set of propositional symbols that are considered to be true at that point). For first-order logic, it's a valuation of the free variables at that point. When one thinks of modelling programs, then that's corresponds to the standard view that the state of an imperative program is the value of all its variables (= state of the memory).

The satisfaction relation $\pi \models \varphi$ is defined inductively over the structure of the formula. We assume that for the formulas of the "underlying" core logic, we have an adequate satisfaction relation $\models_{\mathsf{ul}}$ available, that works on *states*. Note that in case of first-order logic, a signature and its *interpretation* is assumed to be fixed.

Definition 3.2.2 (Satisfaction)**.** A path π satisfies an LTL formula φ, written $\pi \models \varphi$, under the following conditions:

$$\begin{array}{lll}
\pi \models \psi & \text{iff} & \pi_0 \models_{\mathsf{ul}} \psi \text{ with } \psi \text{ from the underlying core language} \\
\pi \models \neg\varphi & \text{iff} & \pi \not\models \varphi \\
\pi \models \varphi_1 \land \varphi_2 & \text{iff} & \pi \models \varphi_1 \text{ and } \pi \models \varphi_2 \\
\pi \models \bigcirc\varphi & \text{iff} & \pi^1 \models \varphi \\
\pi \models \varphi_1 \; U \; \varphi_2 & \text{iff} & \pi^k \models \varphi_2 \text{ for some } k \geq 0, \text{ and} \\
 & & \pi^i \models \varphi_1 \text{ for every } i \text{ such that } 0 \leq i < k
\end{array}$$

The definition of $\models$ covers $\bigcirc$ and U as the *only* temporal operators. It will turn out that these two operators are complete insofar that they can express the remaining operators from the syntax, at least the remaining temporal ones. Those other operators are $\Box$, $\Diamond$, R, and W, according to the syntax we presented earlier. That's a common selection of operators for LTL, but there are sometimes even more added for the sake of convenience and to capture commonly encountered properties a user may wish to express.

We could explain those missing operators as syntactic sugar, showing how they can be macro-exanded into the core operators. What we (additionally) do first is giving a *direct* semantic definition of their satisfaction. As mentioned already earlier, the two important temporal operators "always" and "eventually" are written symbolically like the modal operators necessity and possibility, namely as $\Box$ and $\Diamond$, but their interpretation is slightly different from them. Their semantic definition is straightforward, referring to *all* resp. for *some* future point in time.

The release operator is the dual to the until operator, but is also a kind of "until" only with the roles of the two formulas exchanged. Intuitively, in a formula $\varphi_1\ R\ \varphi_2$, the φ_1 "releases" φ_2's need to hold, i.e., φ_2 has to hold up until and *including* the point where φ_1 first holds and if φ_1 never holds (i.e., never "realeases φ_2), then φ_2 has to hold forever. If there a point where φ_1 is first true and thus releases φ_2, then at that "release point" both φ_1 and φ_2 have to hold. Furthermore, it's a "weak" form of a "reverse until" insofar that it's not required that φ_1 ever releases φ_2.

Definition 3.2.3 (Satisfiability of further operators)**.**

$$\begin{array}{ll} \pi \models \Box\varphi & \text{iff } \pi^k \models \varphi \text{ for all } k \geq 0 \\ \pi \models \Diamond\varphi & \text{iff } \pi^k \models \varphi \text{ for some } k \geq 0 \end{array}$$

$$\begin{array}{ll} \pi \models \varphi_1\ R\ \varphi_2 & \text{iff for every } j \geq 0, \\ & \quad \text{if } \pi^i \not\models \varphi_1 \text{ for every } i < j \text{ then } \pi^j \models \varphi_2 \end{array}$$

$$\pi \models \varphi_1\ W\ \varphi_2 \text{ iff } \pi \models \varphi_1\ U\ \varphi_2 \text{ or } \pi \models \Box\varphi_1$$

Validity and semantic equivalence

Now with the semantics nailed down, we can transport other semantical notions to LTL. Validity, as usual captures "unconditional truth-ness" of a formula. In this case, it thus means, that a formula holds for all paths.

Definition 3.2.4 (Validity and equivalence)**.**

- φ is *(temporally) valid*, written $\models \varphi$, if
$$\pi \models \varphi \text{ for all paths } \pi.$$
- φ_1 and φ_2 are *equivalent*, written $\varphi_1 \sim \varphi_2$, if
$$\models \varphi_1 \leftrightarrow \varphi_2 \text{ (i.e. } \pi \models \varphi_1 \text{ iff } \pi \models \varphi_2\text{, for all } \pi).$$

Example 3.2.5. $\Box$ distributes over $\wedge$, while $\Diamond$ distributes over $\vee$.

$$\begin{array}{c} \Box(\varphi \wedge_1 \varphi_2) \sim (\Box\varphi_1 \wedge \Box\varphi_2) \\ \Diamond(\varphi_1 \vee \varphi_2) \sim (\Diamond\varphi_1 \vee \Diamond\varphi_2) \end{array}$$

In some way, especially from the perspective of model checking, valid formulas are "boring". They express some universal truth, which may be interesting and gives insight to the logics. But a valid formula is also *trivial* in the technical sense in that it does not express any interesting properties. After all, it's equivalent to the formula $\top$. In other words, it's equally useless as a specification as a contradictory formula (one that is equivalent to $\bot$), as it holds for all systems, no matter what.

Valid formulas may still be useful. If one knows that one property implies another (resp. that $\varphi_1 \rightarrow \varphi_2$ is valid), one could model-check using formula φ_1 (which might be easier), and use that to establish that also φ_2 holds for a given model. But still, unlike in logic

$$\begin{tikzcd}
\pi \models \Box p & \bullet_0^p \arrow[r] & \bullet_1^p \arrow[r] & \bullet_2^p \arrow[r] & \bullet_3^p \arrow[r] & \bullet_4^p \arrow[r] & \dots \\
\pi \models \Diamond p & \bullet_0 \arrow[r] & \bullet_1 \arrow[r] & \bullet_2 \arrow[r] & \bullet_3^p \arrow[r] & \bullet_4 \arrow[r] & \dots \\
\pi \models \bigcirc p & \bullet_0 \arrow[r] & \bullet_1^p \arrow[r] & \bullet_2 \arrow[r] & \bullet_3 \arrow[r] & \bullet_4 \arrow[r] & \dots \\
\pi \models p\ U\ q & \bullet_0^p \arrow[r] & \bullet_1^p \arrow[r] & \bullet_2^p \arrow[r] & \bullet_3^{q,(p)} \arrow[r] & \bullet_4 \arrow[r] & \dots \\
\pi \models p\ R\ q & \bullet_0^q \arrow[r] & \bullet_1^q \arrow[r] & \bullet_2^q \arrow[r] & \bullet_3^{p,q} \arrow[r] & \bullet_4 \arrow[r] & \dots \\
\pi \models p\ W\ q & \bullet_0^p \arrow[r] & \bullet_1^p \arrow[r] & \bullet_2^p \arrow[r] & \bullet_3^p \arrow[r] & \bullet_4^p \arrow[r] & \dots
\end{tikzcd}$$

Figure 3.1: Illustration of LTL formulas

and theorem proving, the focus in model checking is not so much on finding methods to derive or infer valid formulas.

The following illustrations are for propositional LTL, where we use p, q and similar for propositional atoms. We also indicate the states by "labelling" the corresponding places in the infinite sequence by mentioning the propositional atoms which are assumed to hold at that point (and leaving out those which are not). However, those are *illustrations*. For instance, when illustrating $\pi \models \bigcirc p$, the illustration shows that p holds at the second point in time (the one indexed with 1). The absence of p for $i = 0$ in the picture is *not* meant to say that it's required that $\neg p$ must hold at $i = 0$ etc. Similar remarks apply to the other pictures.

For the last three examples from Figure 3.1, we should remark the following. For the case of U, the q is required to show up after a **finite** initial sequence of p's. For the R, the sequence of q's can be infinite, and in this case the p does not show up. Also for W, the weak form of until, the sequence of p can be infinite, which in this case is obvious since we have defined $p\ W\ q$ as $p\ U\ q \vee \Box p$.

3.2.3 The Past

The LTL presentation so far focuses on "future" behavior, and the "future" will also be the focus when dealing with alternative logics (like CTL or the μ-calculus). In the section we shortly touch upon switching perspective in that we use LTL to speak about the past; similar switches could be done also for the mentioned other logics, among others. We don't go too deep.

In a way, there is not much new here, if we just talk about the past instead of the future. If we take a transition system (or graph or Kripke structure), in a way it's just "reversing the arrows" (i.e., working with the reverse graph etc.). It corresponds in a way to "run the program in reverse", and then future and past swap their places, obviously. Basically, the same conceptual picture can be done for LTL, considering the linear paths "backwards". Of course, instead of talking about the next state, but backwards (and using reverse paths as models), it's probably clearer if we leave paths as model unchanged, but speak about the *previous* state instead. In general, introduce *past* versions of other temporal operators:

eventually (in the future) becomes sometime earlier in the past, etc. In this way, we can also get a logic which allows to express properties that mix requirements about the future and the past.

It may seem as if the future and the past were basically the same. However, it's actually not true that future and past are 100% symmetric (as we perhaps implied by the above discussion about reversing the perspective). What is asymmetric is the notion of *path*. It is an infinite sequence (or a function $\mathbb{N} \to (P \to \mathbb{B})$, but that's asymmetric insofar it has a start point, but no end. That will require a quite modest variation the way the satisfaction relation $\models$ is defined for the past operators. Apart from that, there is not really much new.

As for the semantics now, we cannot simply use paths, we need pairs (π, j) of paths and positions, where the position indicates the point of "now" [7]. Let's write $\Box^{-1}$, $\Diamond^{-1}$ etc. for **past operators**. The definition of the satisfaction relation is straightforward

$$\begin{array}{lll}(\pi, j) \models \Box^{-1}\varphi & \text{iff} & (\pi, k) \models \varphi \text{ for all } k,\ 0 \le k \le j \\ (\pi, j) \models \Diamond^{-1}\varphi & \text{iff} & (\pi, k) \models \varphi \text{ for some } k,\ 0 \le k \le j\end{array}$$

However, it can be shown that for any formula φ, there is a *future-formula*, a formula without past operators, ψ such that

$$(\pi, 0) \models \varphi \quad \text{iff} \quad (\pi, 0) \models \psi$$

Example 3.2.6 (Past and future LTL). Let's consider as example the property

$$\Box(\varphi \to \Diamond^{-1}\psi) \tag{3.3}$$

mixing future and past operators. It expresses that if a φ occurs at a point, there must have been a point before, where ψ held, and that implication holds always.

That property can be expressed equivalently without past operators, using the (future) release operator:

$$\psi\ R\ (\varphi \to \psi)\ . \tag{3.4}$$

I.e., $(\pi, 0) \models \Box(\varphi \to \Diamond^{-1}\psi)$ iff $(\pi, 0) \models q\ R\ (p \to q)$. $\Box$

Figure 3.2 illustrates the two equivalent formulations of the property.

$$\begin{tikzcd} \bullet^{p\to\Diamond^{-1}q} \arrow[r] & \bullet^{p\to\Diamond^{-1}q} \arrow[r] & \bullet^{p\to\Diamond^{-1}q} \arrow[r] & \bullet^{p\to\Diamond^{-1}q} \arrow[r] & \bullet \arrow[r] & \dots \\ \bullet^{p\to q} \arrow[r] & \bullet^{p\to q} \arrow[r] & \bullet^{p\to q,q} \arrow[r] & \bullet \arrow[r] & \bullet \arrow[r] & \dots \end{tikzcd}$$

Figure 3.2: Illustration of the formulas from Example 3.2.6

3.2.4 Some LTL examples

Let's have a look at a few temporal properties and how they can be captured by LTL. LTL has a formal syntax and semantics, one can capture thus temporal properties unambiguously and precisely.

Often, one starts with informal requirements and it can be difficult to correctly capture informally stated requirements in temporal logic.

Let's try to capture the vaguely formulated property

> "when p then q."

It's not really clear that that is supposed to mean. Here are some more or less plausible formalizations:

$\varphi \to \psi$	$\varphi \to \psi$ holds in the initial state.
$\Box(\varphi \to \psi)$	$\varphi \to \psi$ holds in every state.
$\varphi \to \Diamond\psi$	φ holds in the initial state, ψ will hold in some state.
$\Box(\varphi \to \Diamond\psi)$	(*"response"*)

The last formulation is also called a **response** property (and sometimes one uses a special notation for that, namely $\varphi \rightsquigarrow \psi$). It is not obvious, which one of them (if any) is necessarily what is intended.

Let's do a few more examples.

Example 3.2.7. $\varphi \to \Diamond\psi$: If φ holds initially, then ψ holds eventually.

$$\begin{tikzcd}
\bullet^{\varphi} \arrow[r] & \bullet \arrow[r] & \bullet \arrow[r] & \bullet^{\psi} \arrow[r] & \bullet \arrow[r] & \ldots
\end{tikzcd}$$

This formula will also hold in every path where φ does not hold initially.

$$\begin{tikzcd}
\bullet^{\neg\varphi} \arrow[r] & \bullet \arrow[r] & \bullet \arrow[r] & \bullet \arrow[r] & \bullet \arrow[r] & \ldots
\end{tikzcd}$$

□

Example 3.2.8 (Response). $\Box(\varphi \to \Diamond\psi)$

Every φ-position coincides with or is followed by a ψ-position.

$$\begin{tikzcd}
\bullet \arrow[r] & \bullet^{\varphi} \arrow[r] & \bullet \arrow[r] & \bullet^{\psi} \arrow[r] & \bullet \arrow[r] & \bullet^{\varphi,\psi} \arrow[r] & \ldots
\end{tikzcd}$$

This formula will also hold in every path where φ never holds.

$$\begin{tikzcd}
\bullet^{\neg\varphi} \arrow[r] & \bullet^{\neg\varphi} \arrow[r] & \bullet^{\neg\varphi} \arrow[r] & \bullet^{\neg\varphi} \arrow[r] & \bullet^{\neg\varphi} \arrow[r] & \ldots
\end{tikzcd}$$

□

Example 3.2.9 (∞). $\Box\Diamond\psi$ There are infinitely many ψ-positions.

$$\bullet^{\psi} \longrightarrow \bullet \longrightarrow \bullet \longrightarrow \bullet^{\psi} \longrightarrow \bullet \longrightarrow \bullet^{\psi} \longrightarrow \bullet \longrightarrow \ldots$$

□

Note that this formula can be obtained from the previous one, $\Box(\varphi \to \Diamond\psi)$, by letting $\varphi = \top$: $\Box(\top \to \Diamond\psi)$.

Before we continue with some more examples, let's pause for a while and look at the previous three examples, and compare them with a simpler one, like $\Box p$. It's an example of an *invariant property* or just an invariant, requiring that p always holds.

The previous examples, maybe especially the last one about about infinitely many occurrences of something, feel more complex, not just because there are the formula is bigger. Here we discuss and give arguments what's distinguishes the formulas of the examples from, for instance, the invariance $\Box p$.

Let's assume we have some system and we want to check whether those properties from the examples hold. Let's assume we don't do model checking, but something simpler like *monitoring* or *run-time verification*. That means, we are dealing with a *running* system and we keep an eye on the execution path and the temporal formula to see whether it holds or not. Of course, it's a program, not us, that keeps an eye on the situation, and that's the run-time *monitor*. The task is simpler than model checking insofar run-time verification deals with one execution, whereas model checking attempts to systematically explore all of them. Checking only one execution, one cannot hope for full verification of a program. One cannot even hope to establish that the monitored execution satisfies the property. Why's that? That's because paths are *infinite*. The only thing one can hope for is that the monitor detects a *violation* of the specification, and then flags an alarm or takes corrective actions. That is possible for the an invariance property $\Box p$, but not for the previous examples. For instance, the property from Example 3.2.9 states that there are infinitely points in time where the property hold. Monitoring can neither confirm that, it would take an infinite amount of time, nore can it detect a violation. It's a property one cannot monitor. Technically, that distinction can be captured by classifying properties as safety properties on the one hand, the ones that can be monitored, and *liveness* properties, those that can't.

Obviously, properties can contain both aspects, like in a conjunction of $\Box\varphi$ and one of the three previous examples. However, it can be shown that every LTL formula can be equivalently expressed by a conjunction of a pure safety property and a pure liveness properties. We will talk more precisely about safety vs. liveness later.

The discussion here reasoned that liveness properties like the one from Example 3.2.9 are intuitively more complex, by pointing out that they inuitively cannot be monitored, unlike the safety properties. LTL model checking can check all of LTL, safety and liveness, at least for finite-state systems, but also there, liveness properties are more tricky to deal with. Model-checking safety properties are actually almost a non-brainer. Take the simplest safety property $\Box p$. Given a finite-state system, how does one do that? Just exploring at all possible reachable states, see if any of them violates p. Of so, the property does not hold, if one finds no violation, the property holds. So that's a straightforward graph search. Of coursse the state-graph may be immensely huge. So it requires clever representation techniques and exploration strategies to do that in practice. But the problem itself is a graph search, something one learns in the first or second semester, if not earlier.

It's not obvious at all how one can check liveness properties, like the one from Example 3.2.9 by a finite system. We will see that later, when we address the algorithmic problem to LTL model checking.

Let's continue with some more examples- One can check one's intuition also on the following examples, whether they are safety or liveness properties (or neither).

Example 3.2.10 (Permanence)*.* Eventually φ will hold *permanently:* $\Diamond\Box\varphi$.

$$\begin{tikzcd} \bullet \arrow[r] & \bullet^{\varphi} \arrow[r] & \bullet \arrow[r] & \bullet \arrow[r] & \bullet^{\varphi} \arrow[r] & \bullet^{\varphi} \arrow[r] & \bullet^{\varphi} \arrow[r] & \ldots \end{tikzcd}$$

Equivalently formulated: there are *finitely* many $\neg\varphi$-positions. □

Example 3.2.11*.* $(\neg\varphi)\ W\ \psi$

The first φ-position must coincide or be preceded by a ψ-position.

$$\begin{tikzcd} \bullet^{\neg\varphi} \arrow[r] & \bullet^{\neg\varphi} \arrow[r] & \bullet^{\neg\varphi} \arrow[r] & \bullet^{\psi} \arrow[r] & \bullet^{\varphi} \arrow[r] & \bullet \arrow[r] & \bullet \arrow[r] & \ldots \end{tikzcd}$$

φ may never hold

$$\begin{tikzcd} \bullet^{\neg\varphi} \arrow[r] & \bullet^{\neg\varphi} \arrow[r] & \bullet^{\neg\varphi} \arrow[r] & \bullet^{\neg\varphi} \arrow[r] & \bullet^{\neg\varphi} \arrow[r] & \bullet^{\neg\varphi} \arrow[r] & \bullet^{\neg\varphi} \arrow[r] & \ldots \end{tikzcd}$$

□

Example 3.2.12*.* $\Box(\varphi \to \psi\ W\ \chi)$

Every φ-position initiates a sequence of ψ-positions, and if terminated, by a χ-position.

$$\begin{tikzcd} \bullet \arrow[r] & \bullet^{\varphi,\psi} \arrow[r] & \bullet^{\psi} \arrow[r] & \bullet^{\psi} \arrow[r] & \bullet^{\chi} \arrow[r] & \bullet \arrow[r] & \bullet^{\varphi,\psi} \arrow[r] & \ldots \end{tikzcd}$$

The sequence of ψ-positions need not terminate.

$$\begin{tikzcd} \bullet \arrow[r] & \bullet^{\varphi,\psi} \arrow[r] & \bullet^{\psi} \arrow[r] & \bullet^{\psi} \arrow[r] & \bullet^{\psi} \arrow[r] & \bullet^{\psi} \arrow[r] & \bullet^{\psi} \arrow[r] & \ldots \end{tikzcd}$$

□

Example 3.2.13 (Nested waiting-for). A *nested waiting-for formula* is of the form

$$\Box(\varphi \rightarrow (\psi_m \; W \; (\psi_{m-1} \; W \cdots (\psi_1 \; W \; \psi_0) \cdots))),$$

where $\varphi, \psi_0, \ldots, \psi_m$ in the underlying logic. For convenience, we write

$$\Box(\varphi \rightarrow \psi_m \, W \, \psi_{m-1} \, W \; \cdots \; W \, \psi_1 \, W \, \psi_0).$$

$$\begin{tikzcd}
\ldots \arrow[r] & \bullet^{\varphi,\psi_m} \arrow[r] & \bullet^{\psi_m} \arrow[r, dotted] & \bullet^{\psi_m} \arrow[r] & \bullet^{\psi_{m-1}} \arrow[r, dotted, no head] & \bullet^{\psi_{m-1}} \arrow[r] & \ldots
\end{tikzcd}$$

$$\begin{tikzcd}
\ldots \arrow[r] & \bullet^{\psi_2} \arrow[r, dotted, no head] & \bullet^{\psi_2} \arrow[r] & \bullet^{\psi_1} \arrow[r, dotted, no head] & \bullet^{\psi_1} \arrow[r] & \bullet^{\psi_0} \arrow[r] & \ldots
\end{tikzcd}$$

Every φ-position initiates a succession of intervals, beginning with a ψ_m-interval, ending with a ψ_1-interval and possibly terminated by a ψ_0-position. Each interval may be empty or extend to infinity. □

3.2.5 Dual connectives and complete sets of connectives

In logics, not just in modal logics, one finds often pairs of operators, which are each other's opposites. The two quantifiers $\forall$ and $\exists$ are an example. Opposite does not mean that one is the negation of the other. Clearly $\neg\forall\varphi$ means something else than $\exists\varphi$. But it corresponds to $\exists\neg\varphi$. This form of being in opposition with each other is called *duality*.

Definition 3.2.14 (Duals)**.** For binary boolean connectives $\circ$ and $\bullet$, we say that $\bullet$ is the *dual* of $\circ$ if

$$\neg(\varphi \circ \psi) \sim (\neg\varphi \bullet \neg\psi).$$

Similarly for unary connectives: $\bullet$ is the dual of $\circ$ if $\neg \circ \varphi \sim \bullet\neg\varphi$.

Duality is *symmetric*, i.e., if $\bullet$ is the dual of $\circ$ then $\circ$ is the dual of $\bullet$. Thus we may refer to two connectives as dual to each other.

The $\circ$ and $\bullet$ operators are meant as "placeholders". One can have a corresponding notion of duality for the unary operators $\Diamond$ and $\Box$, and even for null-ary "operators".

Concerning propositional connectives, $\wedge$ and $\vee$ are duals, and $\neg$ is (trivially) its own dual.

$$\neg(\varphi \wedge \psi) \sim (\neg\varphi \vee \neg\psi).$$

If we use $\not\leftarrow$ for the negated reverse implication, then that is the dual of $\rightarrow$?:

$$\begin{aligned}\neg(\varphi \not\leftarrow \psi) &\sim \varphi \leftarrow \psi\\ &\sim \psi \rightarrow \varphi\\ &\sim \neg\varphi \rightarrow \neg\psi\ .\end{aligned}$$

A set of connectives is *complete* (for boolean formulae) if every other connective can be defined in terms of them. Our set of connectives is complete (e.g., $\not\leftarrow$ can be defined), but also subsets of it, so we don't actually need all the connectives.

Example 3.2.15. The set of connectors $\vee$ and $\neg$ is complete. $\wedge$ is the dual of $\vee$, as mentioned. The other connectors can be expressed as follows:

$$\begin{array}{rcl}\varphi \rightarrow \psi & \sim & \neg\varphi \vee \psi\\ \varphi \leftrightarrow \psi & \sim & (\varphi \rightarrow \psi) \wedge (\psi \rightarrow \varphi)\\ \top & \sim & p \vee \neg p\\ \bot & \sim & p \wedge \neg p\end{array}$$

□

Actually, it's not just a complete set, it's also a minimal complete selection in the sense that one cannot remove one of the two connectors without loosing completeness. As is well-known, it's not the only minimal complete choice of operators. An even smaller selection is the set consisting just of NAND (or just of NOR).

In LTL, the two operators $\Box$ and $\Diamond$ are duals:

$$\neg\Box\varphi \sim \Diamond\neg\varphi \quad \text{and} \quad \neg\Diamond\varphi \sim \Box\neg\varphi \tag{3.5}$$

As a side remark: the symbols $\Box$ and $\Diamond$ are also part of modal logics in general, with a different interpretation. But also there, both are duals of each other.

Back to LTL, $\Box$ and $\Diamond$ are not the only temporal duals. Also U and R are each other's duals.

$$\neg(\varphi\ U\ \psi) \sim (\neg\varphi)\ R\ (\neg\psi) \quad \text{and} \quad \neg(\varphi\ R\ \psi) \sim (\neg\varphi)\ U\ (\neg\psi) \tag{3.6}$$

We don't need all our temporal operators either:

Proposition 1 (Complete set of LTL operators)**.** The set of operators $\vee, \neg, U$, and $\bigcirc$ is *complete* for LTL.

Proof. The remaining temporal operators can be expressed as follows:

$$\begin{array}{rcl}\Diamond\varphi & \sim & \top\ U\ \varphi\\ \Box\varphi & \sim & \bot\ R\ \varphi\\ \varphi\ R\ \psi & \sim & \neg(\neg\varphi\ U\ \neg\psi)\\ \varphi\ W\ \psi & \sim & \Box\varphi \vee (\varphi\ U\ \psi)\end{array}$$

The completeness for the propositional connectives has been covered earlier. □

invariant	$\Box\varphi$
example of a liveness property	$\Diamond\varphi$
obligation	$\Box\varphi \lor \Diamond\psi$
recurrence	$\Box\Diamond\varphi$
persistence	$\Diamond\Box\varphi$
reactivity	$\Box\Diamond\varphi \lor \Diamond\Box\psi$

Table 3.2: Classification of LTL properties

3.2.6 Classification of properties

We have seen a couple of examples of specific LTL formulas, i.e., specific properties. Specific "shapes" of formulas are particularly useful or common, and they sometimes get specific names (like "response" or "permanence"). If we take $\bigcirc$ and U as a complete core of LTL, then already the shape $\top\ U\ \varphi$ is so useful that it does not only deserve a special name, it even has a special syntax or symbol, namely $\Diamond$. We have encountered other examples before as well (like *permanence*) and in the following we will list some more.

Another very important classification or characterization of LTL formulas is the distinction between *safety* and *liveness*. Actually, one could see it not so much as a characterization of LTL formulas, but of *properties* (of paths). LTL is a specific notation to describe properties of paths (where a property corresponds to a set of paths). Of course not all sets of paths are expressible in LTL (why not?). The situation is pretty analogous to that of regular expressions and regular languages. Regular expressions play the rule of the syntax and they are interpreted as sets of finite words, i.e., as *properties* of words. Of course not all properties of words, i.e. languages, are in fact regular, there are non-regular languages (context-free languages etc.).

Coming back to the LTL setting: it's better to see the distinction between safety and liveness as a qualification on path *properties* (= sets or languages of infinite sequences of states), but of course, we then see which kind of LTL formulas are capturing a safety property or a liveness property.

Note (again) that "safety" or "liveness" is not property of a paths, it's a property of path properties, so to say. In other words, there will be no LTL formula expressing "safety" (it makes no sense), there are LTL formulas which correspond to a safety property, i.e., expresse a property that belongs to the set of all safety properties.

There is a kind of "duality" between safety and liveness in that safety is like the "opposite" of liveness, but it's not that properties fall exactly into these to categories. There are properties (and thus LTL formulas) that are neither safety properties nor lifeness properties.

Safety properties: never anything bad happens

Table 3.2 contains a few kind of formulas, where φ, ψ are non-temporal formulas. The second one, "eventually φ", is not liveness as such, but it's an example of a liveness property, maybe the simplest one. A similar discussion was done when we drew some parallels between Hoare-logic and dynamic logic.

The *invariant* is a prominent example of a *safety* property. Each invariant property is also a *safety* property. Some even use the words synonymously, but according to the consensus or majority opinion, invariant properties are a subset of safety properties. See for instance the rather authoritative textbook Baier and Katoen [1]. It's however true that *invariants* are perhaps the most typical, easiest, and important form of safety properties and they also represent the essence of them. In particular, if one informally stipulates that safety corresponds to

> "never something bad happens",

then that translates well to an invariant (namely the complete absence of the bad thing: "always not bad"). That characterization of safety is due to Lamport. We also focus on invariance (see Definition 3.2.16), without given the slightly more complex characterization of safety.

As we mentioned earlier, there is a connection to monitoring and run-time verification: Safety-properties are those that can be monitored.

Definition 3.2.16 (Invariant)**.** An *invariant* formula or just invariant is of the form

$$\Box\varphi \tag{3.7}$$

for some propositional (or more generally, non-temporal) formula φ.

Safety formula of the (simplified) form from Definition 3.2.16 express *invariance* of some *state property* φ: that φ holds in every state of the computation. A state property is just what the name implies. It's a property of individual points in time, as opposed to properties of paths. This distinction will be more prominent later in branching time logics like CTL or similar. State properties here are captured by the underlying logic, like propositional logics. Path properties are the the temporal properties.

Example 3.2.17 (Mutual exclusion)*. Mutual exclusion* is a safety property. Let c_i denote that process P_i is executing in the critical section. Then

$$\Box\neg(c_1 \land c_2)$$

expresses that it should always be the case that not both P_1 and P_2 are executing in their respective critical section.

Observe: the negation of a safety formula is a liveness formula; the negation of the formula above is the liveness formula

$$\Diamond(c_1 \land c_2)$$

which expresses that eventually it *is* the case that both P_1 and P_2 are executing in the critical section.

□

Liveness properties

As for safety properties, also here we content ourselves with showing a straightforward formula expressing liveness, not by characterizing liveness as a property of properties.

Definition 3.2.18 (Liveness)**.** A *liveness* formula is of the form

$$\Diamond\varphi \tag{3.8}$$

for some propositional (or more generally, non-temporal) formula φ.

Liveness formulae *guarantee* that some event φ eventually happens: that φ holds in at least one state of the computation.

Connection to Hoare logic After having shed some light on safety and liveness in the context of LTL, let's take a quick detour to one famous program logic, namely Hoare logic. As with most logics, there is not just the one Hoare logic, it's more style of logics with many realization.

The lecture does not formally introduce (a specific variant of a) Hoare logic, we just do some general remarks, to point out a parallel with safety and liveness.

Hoare-logic, named by it's inventor Tony Hoare, is a so-called program logics. It's formulas, ultimately, speak about states of programs, and how that states changes. BTW: since the logic is about talking and reasoning of state changes, it's used for imperative program. To do so, it makes use of *assertions*. That's nothing else than formulas in typically first-order logic (or some variation or fragment) with free variables. Also outside of (Hoare-logic) program verification, assertions are often supported by programming languages. Java, for instance, has a `assert` command. The argument of such a command is an expression of boolean type. That expression of course can contain some variables, and also Boolean connectors. Quantifiers are not supported, so it's a fragment of first-order logic, supporting propositional logic and the possiblity to talk about variables and functions (or methods) and relations. In that programmer's use of assertions, the purpose is not so much to do program verification or model checking, it's run-time assertion checking. If in a run, an assertion at some point in the program code is violated, an exception is raised.

Hoare logics *is* concerned with verification, and a Hoare-logic proof system provide rules that capture the effect of the constructs of the programming language on the program state, when the construct is executed.

That is done by relating the "assertion" before a statement with the one afterwards. This leads to the well-known pre-condition and post-condition formulation, typical for Hoare logic., especially appropriate for sequential prog

There are two ways a Hoare-logics specification on a program can be interpreted. The two ways are called *partial correctness* and *total correctness*. The first one states, that when starting in a state that satisfies the pre-condition, should the program terminate, then it will be in a state that satisfies the post-condition. Note that under the partial correctness interpretation, a pre- and post-condition specification has no opinion on non-terminating programs.

That's different for total correctness. Total correctness states that, when starting in a state that satisfies the precondition, then it's guaranteed that the program terminates, and, upon termination, satisfies the post-condition.

Of we use *terminated* as predicate to express termination, we can express partial and total correctness of a program P with pre-condition φ and post-condition ψ as follows

$$\varphi \rightarrow \Box(\mathit{terminated}(P) \rightarrow \psi) \quad \text{resp.} \quad \varphi \rightarrow \Diamond(\mathit{terminated}(P) \land \psi). \tag{3.9}$$

The formulas are formulas in LTL notation, in Hoare logic would express pre- and post-conditions notationally as $\{ \varphi \}\ P\ \{ \psi \}$ (a so-called Hoare-triple of pre-condition, post-condition and the program in the middle.)

The two formulas of equation (3.9) are not of the form called safety and liveness from Definitions 3.2.16 and 3.2.18. The form is sometimes called *conditional* safety and *conditional* liveness formula, being conditioned on the precondition φ.

When independent from the precondition, resp. assume φ to be "true", partial and total correctness are directly safety and liveness conditions of the form introduced.

Partial correctness may look as a weaker version of total corrctness, already the name seems to imply that. Perhaps surprisingly it turns out partial and total correctness are dual to each other. Let's focus on the *un*-conditional version, post-condition only. Then, it should become not so surprising after all: $\Box$ and $\Diamond$ are defined refering to *all* time points in the future, resp. to *some* time point in the future. And $\forall$ and $\exists$ are dual operators.

Let's introduce the following ammbreviations:

$$PC(\psi) \triangleq \Box(\mathit{terminated} \rightarrow \psi) \quad \text{and} \quad TC(\psi) \triangleq \Diamond(\mathit{terminated} \land \psi)$$

Then

$$\neg PC(\psi) \sim PC(\neg\psi) \quad \text{and} \quad \neg TC(\psi) \sim TC(\neg\psi)$$

Other classes of formulas: Obligation, recurrence and persistence, reactivity

Here the definition of obligations.

Definition 3.2.19 (Obligation). A *simple obligation* formula is of the form

$$\Box\varphi \lor \Diamond\psi$$

for propositional (or generally non-temporal) formulas φ and ψ.

Such a property can equivalently be written as

$$\Diamond\varphi \rightarrow \Diamond\psi$$

The equivalent form $\Diamond\chi \rightarrow \Diamond\psi$ states that if some state satisfies χ, then some state must satisfy ψ.

Obligations subsume safety and liveness properties (and the form presented here).

Proposition 2. Every safety and liveness formula is also an obligation formula.

Proof. It's a consequence of the following equivalences.

$$\Box\varphi \sim \Box\varphi \vee \Diamond\bot \quad \text{and} \quad \Diamond\varphi \sim \Box\bot \vee \Diamond\varphi$$

and the facts that $\models \neg\Box\bot$ and $\models \neg\Diamond\bot$. □

To be recurrent means to occur over and over again. That can be caputured as follows.

Definition 3.2.20 (Recurrence)**.** A *recurrence* formula is of the form

$$\Box\Diamond\varphi$$

for some propositional (or more generally, non-temporal) formula φ.

It implies that there are *infinitely many* positions where φ holds. A *response* formula, of the form $\Box(\varphi \to \Diamond\psi)$, is equivalent to a recurrence formula, of the form $\Box\Diamond\psi$, if we allow χ to be a past-formula.

$$\Box(\varphi \to \Diamond\psi) \leftrightarrow \Box\Diamond(\neg\varphi)\ W^{-1}\ \psi$$

Next we express some form of fairness as a recurrence property. There are two main variants of fairness, weak and strong fairness. Generally it means that given two or more processes competing repeatedly on some resource, it's not that case that one of the competing processes is neglected all the time. If applying long enough or often enough for a resource must ultimately give access to the resource. Resource can mean various things, typical is processor time or acquiring a lock and getting access to a critical region. Equation (3.10) below is formulated referring to a step τ, that is *enabled* resp. is taken.

Example 3.2.21 (Weak fairness)*.* Weak fairness can be specified as the following recurrence formula.

$$\Box\Diamond(\mathit{enabled}(\tau) \to \mathit{taken}(\tau)) \tag{3.10}$$

□

An equivalent form is

$$\Box(\Box\mathit{enabled}(\tau) \to \Diamond\mathit{taken}(\tau)),$$

Fairness is a concept from concurrent or parallel systems, and can be expressed in LTL. Later we will see, fairness is *not* expressible in CTL, another important temporal logics.

As said, fairness means, that some actions or processes are not "unduly neglected" in favor of other actions or processes. To speak of fairness, there must be an element of choosing from alternatives (at least two). Another element is that the choice is *repeated.* If one has just one choice between two alternatives a *one* point in time and one is chosen, then that is neither fair nor unfair. However, if one *repeatedly* favors one alternative over the other, then that is unfair. Repeatedly refers actually to *infinitely often.* So just selecting alternative A 500 times before choosing B is not a violation of fairness, *always* in an infinite run choosing A is.

Often, the choices being made is between actions of different processes, choosing to execute a statement of process P_1 or of P_2 (if one has a system of 2 processes). So, that makes fairness a (typically desired) property of a scheduler. It's also a very “abstract” and general notion (taken an “infinitely long” perspective before one speaks about fairness or unfairness). As hinted at above, if a scheduler assigns 5 times as many slots to P_1 compared to P_2, thus executing it five times as fast as the other, one might see that as “unfair” in some sense, but not in the technical sense of fairness. There are also “bounded” versions of fairness (which we don't treat here)

Fairness, however, typically is refined insofar that one distinguishes *strong* and *weak* fairness. It's connected with scheduling as well and the notion of *enabledness* (of transitions or steps). Transitions can be enabled or not. That in particular applies to actions in connection with *synchronization*. For instance, if a process is at a point where the next action is to take a lock, that step may or may not be enabled, depending on whether the lock is free or taken. An non-enabled action that is never scheduled does not count as unfair scheduling: it's not the scheduler's fault that, for instance, the owner of the lock does not give it back, thereby enabling the other processes waiting on the lock. It may be characterized as unfairness at some higher-level of looking at the program or characterized as different form of defect (maybe caused by a deadlock), but it's not unfairness on the level of scheduling actions: only actions that are enabled and could thereby be actually selected count when thinking about fairness.

Weak fairness means, an enabled action cannot *remain enabled* forever without being chosen. Strong fairness requires that an action cannot be *not chosen* if it is enabled *infinitely often* (but not necessarily continuously enabled). For instance, in the lock-illustration: if the lock is taken and released by other processes, then for some process waiting to take it, the corresponding action *toggles* between being enabled and disabled. *Weak* fairness does not require that the process ultimately get's the lock, but *strong* fairness would ensure that.

Weak and strong fairness are be “recurrent” (sorry for the pun) themes in dealing with concurrent system. They may also show up in later parts of the lecture, and we will also see how to express the weak variant in LTL later.

The next property we look at is called *persistence* or alternatively *stabilization.*

Definition 3.2.22 (Persistence)**.** A *persistence* formula is of the form

$$\Diamond\Box\varphi$$

for some propositional (or more generally non-temporal) formula φ.

Persistence of φ means, that at some point onwards, it will from then on always φ. This is another way of saying that only *finitely* many position satisfy $\neg\varphi$. In other words, persistance is dual to the property “infinitely often”, which is also called recurrence. Recurrence and persistence are duals:

$$\neg(\Box\Diamond\varphi) \sim (\Diamond\Box\neg\varphi) \quad \text{and} \quad \neg(\Diamond\Box\varphi) \sim (\Box\Diamond\neg\varphi)$$

The last class of properties is called *reactivity*, combining persistence and recurrence.

Definition 3.2.23 (Reactivity)**.** A *simple reactivity* formula is of the form

$$\Box\Diamond\varphi \vee \Diamond\Box\psi \tag{3.11}$$

for propositional (or more generally, non-temporal) formulas φ and ψ.

A very general class of formulas are conjunctions of reactivity formulae.

An equivalent formulation of the reactivity formula is

$$\Box\Diamond\psi' \rightarrow \Box\Diamond\varphi,$$

(or also $\Box(\Box\Diamond\chi \rightarrow \Diamond\psi)$) which states that if the computation contains infinitely many χ-positions, it must also contain infinitely many ψ-positions.

3.2.7 GCD Example

Let's look at a small example, a really small one. It mainly intended to illustrate LTL one more time, connecting it to the behavior of a program. It's not a typical program for LTL problems or model checking, insofar that it's effectively a sequential program.

Program: GCD

```
        $P$ :: [ in $a, b$ : integer where $a > 0, b > 0$ ;
                 local $x, y$ : integer where $x = a, y = b$ ;
                 out $g$ : integer ;
    $P_1$ :: [ $l_0$ : [ $l_1$ : while $x \neq y$ do $l_2$ : [
                     [ $l_3$ : await $x > y$ ; $l_4$ : $x := x - y$ ; ]
                     or
                     [ $l_5$ : await $y > x$ ; $l_6$ : $y := y - x$ ; ] ]
                 $l_7$ : $g := x$ ; $l_8$ : ] ] ]
```

The code is some form of pseudo-code for concurrent processes. The code should be roughly understandable. The body of the process is the lower half of the code. Basically a big while-loop, consisting of two alternatives, each consisting by of an `await`-statement. What exatcly that is does not matter right now. Effectively, in this example the body of the loop is a two armed alternative. At no point, both branches of the alternative are enabled, as at most one condition guarding the branches can be true. Additionally the branches are executed *atomically*, and that effectively makes the program sequential. So it's a fancy version of writing the good old GCD algorithm.

The variables a and b are the *inputs* of the program, and g is the output. As is often assumed and good practice, the input variables are immutable, and the output is likewise not mutated, up until the end when the result is to be returned.

The red l's that show up in the program are *not* part of the program. They are indicated to refer to different control-flow points in the code (and l stands for location or label). One can (and should) make a difference between labels in the program and locations in the following sense. In the program, there are labels that refer semantically to the same control-flow points. For instance l_0 and l_1. So while in the syntax of the program code, l_0 and l_1 are different labels, it's best to think of them to refer to the same location. We don't make that formal nor do we make explicit how to turn a program into a control-flow graph; the locations correspond to nodes in a control flow graph. Control-flow graphs are common intermediate representations inside a compiler, and actually also model checkers like Spin use control-flow graph internally. After all, Spin compiles a programming-language notation for an imperative, concurrent language to executable code, which executes not just the code, but runs the code being checked against the LTL formula and arranging for exploring the state-space (plus doing a lot of further tricks and optimizations).

Below is a computation π of our recurring GCD program. States are of the form $\langle l, x, y, g\rangle$, consisting of the location and the values of the local variables x and y as well as the variable g for the result. One can make the argument, that also the values of a and b are part of the state, but those don't change, so we can safely ignore them as uninteresting.

States are of the form $\langle l, x, y, g\rangle$, and the execution shown below is an (infinite) sequence of states. It's infinite, since for LTL, we need to work with infinite paths (which arise form the infinite execution). The GCD program is terminating, so it does not technically continues executing forever, but the usual "trick" to handle that is obvious: when the program terminates, one assumes that it continues doing steps without changes to the status. This is known as (one form of) *stuttering*, and we will cover aspects of stuttering later (but, as said, the idea as such is rather obvious).

Let $at(l_n)$ represent the formula expressing that the program is at location l_n, i.e., at a state of the form $\langle l_n, \_, \_, \_\rangle$. Let furthermore *terminated* represent the formula $at(l_8)$.

$$\begin{array}{llll}
\pi: & \langle l_1, 21, 49, 0\rangle \to \langle l_2^b, 21, 49, 0\rangle \to \langle l_6, 21, 49, 0\rangle \to \\
& \langle l_1, 21, 28, 0\rangle \to \langle l_2^b, 21, 28, 0\rangle \to \langle l_6, 21, 28, 0\rangle \to \\
& \langle l_1, 21, 7, 0\rangle \to \langle l_2^a, 21, 7, 0\rangle \to \langle l_4, 21, 7, 0\rangle \to \\
& \langle l_1, 14, 7, 0\rangle \to \langle l_2^a, 14, 7, 0\rangle \to \langle l_4, 14, 7, 0\rangle \to \\
& \langle l_1, 7, 7, 0\rangle \to \langle l_7, 7, 7, 0\rangle \to \langle l_8, 7, 7, 7\rangle \to \cdots
\end{array}$$

Now, do the following properties hold for π? And why?

$$\begin{array}{ll}
\Box \mathit{terminated} & \text{(safety)} \\
at(l_1) \to \mathit{terminated} & \\
at(l_8) \to \mathit{terminated} & \\
at(l_7) \to \Diamond \mathit{terminated} & \text{(conditional liveness)} \\
\Diamond at(l_7) \to \Diamond \mathit{terminated} & \text{(obligation)} \\
\Box(\mathrm{gcd}(x, y) \doteq \mathrm{gcd}(a, b)) & \text{(safety)} \\
\Diamond \mathit{terminated} & \text{(liveness)} \\
\Diamond\Box(y \doteq \mathrm{gcd}(a, b)) & \text{(persistence)} \\
\Box\Diamond \mathit{terminated} & \text{(recurrence)}
\end{array}$$

3.2.8 Exercises

Exercises

1. Show that the following formulas are (not) LTL-valid.
   a) $\Box\varphi \leftrightarrow \Box\Box\varphi$
   b) $\Diamond\varphi \leftrightarrow \Diamond\Diamond\varphi$
   c) $\neg\Box\varphi \rightarrow \Box\neg\Box\varphi$
   d) $\Box(\Box\varphi \rightarrow \psi) \rightarrow \Box(\Box\psi \rightarrow \varphi)$
   e) $\Box(\Box\varphi \rightarrow \psi) \vee \Box(\Box\psi \rightarrow \varphi)$
   f) $\Box\Diamond\Box\varphi \rightarrow \Diamond\Box\varphi$
   g) $\Box\Diamond\varphi \leftrightarrow \Box\Diamond\Box\Diamond\varphi$
2. A *modality* is a sequence of $\neg$, $\Box$ and $\Diamond$, including the empty sequence ϵ. Two modalities π and τ are *equivalent* if $\pi\varphi \leftrightarrow \tau\varphi$ is valid.
   a) Which are the non-equivalent modalities in LTL, and
   b) what are their relationship (ie. implication-wise)?

3.3 Automata-based model checking

3.3.1 LTL model checking

Given a formula φ in some logic and a model M, appropriate for the logic, *model-checking* in general is about answering the question whether

$$M \models^? \varphi, \tag{3.12}$$

i.e., whether the model satisfies the formula. In our case of temporal logics, the model takes the form of a transition system together with a starting state, there the transitions represent steps of a program or system (resp. perhaps an abstraction of a real system).

That's a straight enough problem, but how to address it depends on the concrete form of the models and the concrete logics.

As far as the models go, and independent from the logics, a big challenge is typically the size of the model or transition system: the **state space** of the model is simply *huge* for realistic problems, maybe even infinite. At least in principle infinite: if one has programs that work with numbers, then there are infinitely many. One can make the argument that on a computer, there are only finitely many representable numbers, like up-to `MAXINT`, but even if in this case the state space is actually finite, it's "practically infinite", since it's just too many. Similarly if one tries to verify programs with *dynamic* data structures, like lists and trees etc. Also there, the argument that computer memory is limited and this there is an upper bound on the size of the trees may be technically true, but not really helpful in practice.

Of ourse, not all problems require dynamic data structures, some programs run on firmware or hardware, and indeed, model checking hardware-close systems and algorithms is seen as one important application area; software being way more challenging.

A prime application area for model checking is also concurrent systems (hardware or software). When dealing with natural numbers and data structures, which cause huge or infinite state spaces, other (or additional) techniques may be more adequate, used in combination with model checking. For concurrent system, there is another reason why state-spaces become huge. That's because of processes running concurrently can be executed with many different interleavings. What makes it tricky is that concurrency-related bugs are often ...

3.3.2 Automata-based LTL model checking

As said, there are many variations on the theme of temporal logic model checking: different logics, different kinds of target systems, different models, different techniques, different optimizations, ... We focus in this section on one prominent technique for LTL, **automata-based** model checking.

It's a method for *finite-state* models and it an example of so-called *explicit* state-space model checking. It's called *explicit* state model checking, as the states of the system are individually represented and the model-checking processe explores the state-space by one individual state after another. An alternative, which represents *sets* of states jointly, is known as *symbolic* model checking (but that's for later).

Let's assume, the transition systems come with one specific state, the initial state. Kripke frames don't come equipped with a start state, but for model checking transition systems representing programs, it's conventional to assume one specific state. It actually does not change the problem in the least, but streamlines the following discussions a bit, by avoiding that we have to say "assume a Kripke-structure M together with a state s..." over an over again. And as said, it's conventional for model checking systems anyway. So, a model M, a transition system, is now a defined like what we introduced as Kripke structure but with additionally an initial state.

With that triviality out of the way, let's ask ourselves: in the setting of LTL, with transition systems as models, and given the model-checking problem from equation (3.12), how could one go for it?

For simple LTL-formulas like invariance properties $\Box p$, it's conceptually pretty simple. We are given the model as a transition system, for each state we have the valuation for the propositional atoms. In particular we are given the status of p for each place. The invariance from equation (3.12) holds, if all paths in M, that start from its initial state s_0, satisfy p (see Definition 3.2.2). That's in general infinitely many paths, and each path itself is infinite.

But this infinity is not a big deal: one simply has to explore the transition system, starting from s_0, checking all states *reachable* from s_0. If all reachable states satisfy p, the property holds for all paths starting at s_0, i.e., $M \models \Box p$ holds. If the exploration discovers one place that violates p, then the model-checking question is answered negatively, and one can stop the exploration (unless one is interested in all places that violate p). As we assume that the system is finite, the problem is solved. There may be practical challenges, like the size of the transition system, but *deciding* whether $\Box p$ holds in a finite-state model is conceptually simple.

Of course, $\Box p$ is arguably the simplest temporal property, an easy instance of a safety property. Things don't look quite so easy with more complex properties. Let's take $\Diamond\Box p$ as example, expressing *infinitely many occurrences* of p on all path. How to check that? If we were interested in checking, that there *exists* a path in M, starting from s_0, that contains infinitely many places where p holds, one could come up with the following: starting from s_0, try to reach a state s such that, starting from s, one finds a *cycle* in the graph, leading back to s such that on the cycle, there is a s' where p holds. One could arrange that more efficiently than going on a loop-finding mission for all reachable s's one after the other, but doing those loop-searches naively would already solve the problem. Of course to *decide* the question, one needs to explore, for each place s, all loops. Of course if one has run through one particular cycle one time, one does not have to run through the same cycle again, so one can restrict the seach to simple cycles and that makes the problem finite, so it seems doable.

We made plausible how to look for *some* path that satisfies $\Box p$. The standard form of LTL are not looking whether there exists a path, it is about checking *all* paths. One could try to address that similarly, checking more or less cleverly for reachable states that are contained in cycles in the graph.

All that may be plausible but we need a *general method* that works for all of LTL. That's the main topic of this chapter, automata-based LTL model checking. We start by laying out the general ideas behind the approach, details will come later.

The big picture: automata-based LTL model checking as a form of refutation

In a bird's eye view and very generally, the method can be seen as a **refutation** proof method. In the chapter covering non-temporal logics, we touched upon proof-systems, sketching rule-systems to derive *valid* formulas. Proof by refutation works differently, so the proof systems we sketched back then are *no* refutation systems. Refutation means doing a proof by *contradiction*. Instead of proving a formula φ, one assumes the opposite $\neg\varphi$ and if, assuming the opposite, one is able to derive absurdity $\bot$ (a contradiction), then the original formula φ must be true (or valid etc). In non-intuitionistic logics, that works fine: instead of trying to establish validity of φ, one tries *non-satisfiability* of φ, as both is equivalent. Formulaically:

$$\models \varphi \qquad \text{iff} \qquad \neg\varphi \models \bot\ . \tag{3.13}$$

Remember, that the *core* of the satisfaction relation for LTL is defined as a relation between one paths and one formulas (see Definition 3.2.2). The satisfaction relation between transition systems and formulas from equation (3.12) is *derived* from that, posing a requirement on *all* paths of M starting its initial state.

Formulas describe sets of paths, namely all paths that make the formula true, but likewise transitition systems describes sets of paths, namely all the paths that start in its initial state. We can therefore view the model checking problem $M \models^? \varphi$ from equation (3.12) as entailment problem.

Let's treat the transition system M as specification of all its paths. We can write $[\![M]\!]$ for the set of all its paths starting at its initial state. While going down that road, we might as well say "a path π satisfies M", i.e. writing $\pi \models M$ for $\pi \in [\![M]\!]$. With this picture in mind, then the model-checking question $M \models^{?} \varphi$ is the logical entailment: every path "satisfying" M also satisfies φ.

That does not help much in practically addressing the model checking problem, but it helps to see the analogy to refutation methods. In analogy to validity from the equivalence from equation (3.13), the entailment can be refuted based on the following equivalence:

$$\Gamma \models \varphi \qquad \text{iff} \qquad \Gamma, \neg\varphi \models \bot \; . \tag{3.14}$$

The set of formulas $\Gamma, \neg\varphi$ on the left hand side of $\models$ is interpreted as *conjunction* of the formulas of Γ and $\neg\varphi$. An entailment is refuted if there is not interpretation that satisfies all formulas from Γ and $\neg\varphi$ at the same time. In our picture seeing M as well as φ as specifying a set of paths, we could write

$$M \models \varphi \qquad \text{iff} \qquad M \land \neg\varphi \models \bot \; . \tag{3.15}$$

Alternatively we can formulate entailment as subset requirement:

$$[\![M]\!] \subseteq [\![\varphi]\!] \qquad \text{iff} \qquad [\![M]\!] \cap \overline{[\![\varphi]\!]} = \emptyset \; . \tag{3.16}$$

where we use $\overline{A}$ to represent the complement of a set A. So $\overline{[\![\varphi]\!]}$ is represents the negated formula $[\![\neg\varphi]\!]$.

The big picture (2): steps of the construction

We have seen now that model checking can (also) be understood as refutation problem. And that picture is underlying the automata-based model checking approach which is the topic of this chapter. But it's still just a picture. What we need is to operationalize it, to turn in into an algorithm. So how can one solve the problem

$$[\![M]\!] \cap \overline{[\![\varphi]\!]} = \emptyset \; , \tag{3.17}$$

where M is a transition systems representing the system we want to check and φ the specification in LTL?

In the refutation-discussion we said, the M can also seen as a "logical" specification, namely specifying a set of paths. But of course M and φ are specifications of paths of quite different formalisms. M is a transition system and φ a logical formula. So a notation like $\varphi \land M$ we allowed ourselves (in equation (3.15)) is a bit dubious (but of course the $\land$ is un-dubiously explained by the intersection formulation from equation (3.16)).

To check entailment resp. the refutation formulation from equation (3.16) directly is problematic (in that it does not work..). What we would have to do is for checking the entailment is to calculate two infinite sets of infinite paths and then check that one use *included* in the other. For the refutation formulation, we need do to

1. calculate two infinite sets of infinite paths, $[\![M]\!]$ and $[\![\varphi]\!]$
2. calculate the complement of the latter,
3. calculate the intersection between then,
4. check if the intersection is empty.

Of course, when done directly, so those steps all involve infinite sets of infinite entities and that's not how to do it.

What we need instead is a **finite representation** of those mentioned infinite sets and then doing the steps not on the infinite sets but manipulating their finite representation. Fortunately, we have those finite representations already! Namely the transition system M and the formula φ.

Unfortunatly, though, they are not "on the same level", one is a transition system and the other is an LTL formula. What they have in common is that both describe infinite sets of paths.

To put them on a common level, we have to translate the formula φ to a "transition system". The above steps of the approach include actually one more, that said translation. Actually, one of the steps from above listed 4 can be (and is) omitted. That's step 2), the complementation. One does not need to figure out how to complement a transition system: if one has figured out a way to translate all LTL formulas to a transition system, then one simply translates $\neg\varphi$ to avoid complementation. So the steps, now on the finitite representations of the infinite sets of paths are

1. translate $\neg\varphi$ to $A_{\neg\varphi}$, a finite respresentation with nodes and edges,
2. calculate a representation that corresponds to the intersection, and
3. check if the intersection represents the empty set.

The above sketch of the approach has to be taken with a grain of salt. We said that one step put the $\neg\varphi$ at the same level with M by translating it to a transition system. In principle that's correct, but there is fine-print. The fine print is that the translated $A_{\neg\varphi}$ is not exactly of the same form as the transition system M. We will see that when looking in detail at the construction (but gloss over the details now, when discussing the big pictue). Indeed $A_{\neg\varphi}$ is not called transition system but **automaton**. That's why the approach is called **automata-based LTL model checking**.

The difference between the definition of transition systems and the automata is not big. For instance, we consider transition systems in a form where the *nodes* of the transition system carry information or a labelled. For the automata, it's the *edges* which are labelled (as is standard for automata). Also, the automata have *accepting states*, again standard for automata, whereas M, representing a program or system, does not. In the construction later, one has to take care that one representation is transition labelled and the other one is action labelled, but it's a detail indeed. Both formats, edge-labeled and node-labelled, are interchangable, and the construction later will contain a small step, where M is massaged into a edge-labelled representation, to make A and the system representation to be really on the same level, being two automata not only in spirit, but for real. But that's for later, we ignore that in the rest here, we consider transition systems and automata as basically the same formalism.

Talking about automata and sets of sequences may ring a bell: a very well-known concept are **regular languages**. In that context, a set of sequences is called a **language** and sequences (over an alphabet) are called **words**. Languages are typically infinite sets, finite languages are trivial. It's a well–known fact that regular languages can be described finitely in two different and equivalent ways, by **regular expressions** and by **finite-state automata**.

The paralell is pretty close: we are interested in representing infinite sets of words, LTL corresponds conceptually to regular expressions and in both cases there is automaton representation as a different finite reprentation.

There are important differences as well. The most important one is that here we are dealing with infinite sets of **infinite words** (we call them paths, but that's just a name). Furthermore LTL is quite more expressive (already from the fact that it can express properties about infinite words). Indeed that's a crucial difference. It has to do with the fact that LTL can express liveness properties, whereas regular expressions in a way can only express safety properties. Finally, traditional finite-state automata are defined in such a way that they describe or accept only finite words. For LTL we need to adapt that, so that the automata, still finite-state , accepts infinite words. One way of doing that actually does not even change the definition of finite-state automaton at all, one does just change the conditions under which a word is *accepted* by an automaton. Roughly like that: a standard automaton accepts a word when hitting an accepting state. A **infinte-word automaton** accepts a word if it hits accepting states infinitely often. **Büchi-automaton** is a well-known version of an infinite word automaton, and that's the kind of automaton used for LTL model checking. There are other such automata as well, some equivalent to Büchi-automata, some not, but we just cover the Büchi-flavor which works well for LTL.

So, are we done then? Not quite. Having a finite representation of infinite sets of infinite words in the form of automata is great. But the automata have become quite more expressive thanks to the more complex acceeptance condition. One can implement such an automaton straightforwardly, in the same way that one can implement a standard finite-state automaton, actually since only the acceptences condition has changed, the implementation is unchanged except when to stop generating or acceting a word.

Indeed, since acceptnance requires to hit an accepting state infinitely often, such automata cannot be used to actually accept infinite words (in the way an ordinary finite-state automation can be used to accept finite words, maybe in a lexer). Indeed, using an automaton or something else, how can one expect to check properties of infinite words? It's related to an earlier discussion about safety vs. liveness, and about run-time verification. When dealing with infinite words or runs and their properties, the best one can do is spotting in a finite prefix violations of *safety* properties, liveness properties cannot be monitored or checked via run-time verification.

That sounds like bad news for model checking, but model checking is not about checking individual runs or individual infinite words, it's about working with their automata reprentations.

So what actually needs to be solved algorithmically is

translation: find a translation for LTL formulas to an equivalent Büchi automaton

intersection: construct and automata the represents the intersection of two automata

emptyness: check if the language of a given automata is empty

As said, a construction covering complementation can be avoided, we simply translate $\neg\varphi$.

We address the constructions in detail later, but before we do that, let's elaborate on the parallel between finite-word languages and infinite-word languages and their finite representations, looking also the constructions just mentioned.

Comparison to finite-state automata and regular languages

As said, one can draw a parallel between standard finite-state automata and regular expression on the one hand and Büchi-automata (which are also finite state) and LTL on the other hand. Despites many similarities, the LTL-setting get quite more challenging. And indeed, not all

Emptyness-checking

Determinism vs. non-determinism

Complementation

Automata-level vs. declarative level

3.4 Automata and logic

After having shed light on the concepts behind LTL model checking, it's time to get more technical. In this section we cover the standard finite-state automata, and afterwards Büchi-automata, a well-known representative of finite-state automata for infinite words.

3.4.1 Finite state automata

Let's start by introducing the well-known concept of finite-state automata

Definition 3.4.1 (Finite-state automaton)**.** A *finite-state automaton* is a quintuple $(Q, q_0, , \Sigma, F, \rightarrow)$, where

- Q is a finite set of states
- $q_0 \in Q$ is a distinguished initial state
- the "alphabet" Σ is a finite set of labels (symbols)
- $F \subseteq Q$ is the (possibly empty) set of final states
- $\rightarrow \subseteq Q \times \Sigma \times Q$ is the transition relation, connecting states in Q.

What we call *alphabet* here (with a symbol Σ) is sometimes also called *label set* (maybe with symbol L) and sometimes the elements are also called *actions*. The terminology "alphabet" comes from seeing automata to define words and languages, the word "action" more when seeing the automaton as a system model that represents an (abstraction of) a program.

The notion of *finite state automata* is probably known from elsewhere. It's used directly or in variations in many different contexts. Even in its more basic forms, the concept is known under different names or abbreviations (FSA and NFA, finite automaton, finite-state machine). Minor and irrelevant variations concern details like whether one has one initial state or allows a set of initial states. Sometimes the name is also used "generically", for example, automata which carry more information than just labels on the transitions. For instance, information which is interpreted as input and output on the states and/or the transitions (also known Moore or Mealy machines). Such and similar variations are no longer *insignificant* deviations like the question whether one has one initial state or potentially a set. Nonetheless those variations are sometimes also referred to as FSAs, even if technically, they deviate in some more or less significant aspect from the vanilla definition given here. They are called finite-state machines or finite-state automata simply because they are state-based formalisms with a finite amount of states and some form of transition relation in between (and potentially labelled or interpreted in some particular way or with additional structuring principles).

Other names for related concepts is that of a (finite-state) *transition system*. And even Kripke structures or Kripke models can be seen as a variation of the theme, though in a more logical or philosphical context, the *edges* betwen the workds may not be viewed as *transitions* or operational steps in a evolving system. In Baier and Katoen [1], they call Kripke structure *transition systems* (actually without even mentioning Kripke structures).

We are not obsessed with terminology. But as preview for later: In the central construction about model checking LTL, the system on the one hand will represented as a (finite) transition system where the *states* are labelled and the LTL formula on the other hand will be represented by an *automaton* whose *transitions* are labelled. The automaton will be called *Büchi*-automaton. The definition corresponds to the one just given in Definition **??**. What makes it "Büchi" is not the form or data structure of the automaton itself, it the *acceptance* condition, i.e., the intepretation of the set of accepting states.

Let's illustrate the definition on a small example

Example 3.4.2. The automaton is given by the 6-letter alphabet (or label set) $\Sigma = \{a_0, a_1, \ldots, a_5\}$, by the 5 states q_0, q_2, ..., q_4, with initial state and one final state and the transitions as given in the figure. "Technically", one could enumerate the transitions by listing them as triples or labelled edges one by one, like

$$\rightarrow \; = \; \{(q_0, a_0, q_1), \ldots, (q_2, a_5, q_4)\} \subseteq Q \times \Sigma \times Q \ ,$$

but it does not make it more "formal" nor does it add clarity.

Renaming the letters of the alphabet, the above automaton may be interpreted as a *process scheduler*:

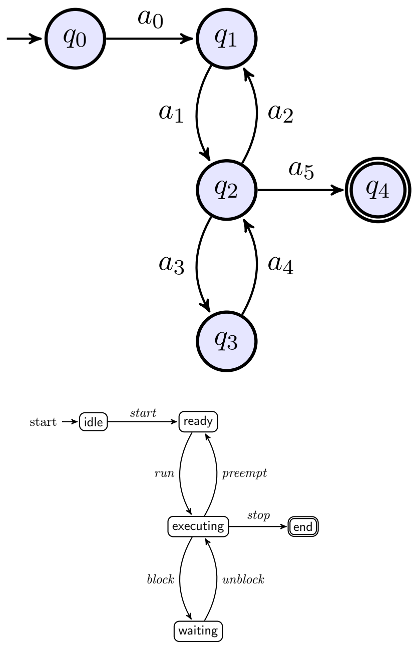


Figure 3.3: FSA scheduler

Determinism vs. non-determinism

Determinism in a system geneally means that, in a given situation or state, the next state is determined. Functions are deterministic: given an input, the function output is determined as well. In that sense, relations can be seen non-deterministic “functions”. For automata and related formalisms, determinism means, being in a state and given a letter from the alphabet, there are at most one successor states. The automata from Definition 3.4.1 can be non-deterministic, the definition is based on a transition *relation*.

Definition 3.4.3 (Determinism)**.** A finite state automaton $\mathcal{A} = (Q, q_0, \Sigma, F, \rightarrow)$ is *deterministic* iff

$$q_0 \xrightarrow{a} q_1 \ \wedge q_0 \xrightarrow{a} q_2 \implies q_1 = q_2$$

for all q_0, q_1, and q_2 from Q and all a in Σ.

The definition of *deterministic* automaton is not 100% equivalent with requiring that there is a transition *function* that, for each state and for each symbol of the alphabet, yields *the* unique successor state. Our definition basically requires that there is *at most* one successor state (by stipulating that, if there are two successor states, they are identical). That means, the successor state, if it exists, is defined by a *partial* transition function.

Sometimes, the terminology of *deterministic* finite-state automaton *also* includes the requirement of *totality*, i.e., the transition relation is a total relation, wich makes it a *total function.*

I.e., the destination state of a transition is uniquely determined by the source state *and* the transition label. An automaton is called **non-deterministic** if it does not have this property. We prefer to separate the issue of deterministic reaction to an input in a given states ("no two different outcomes") from the issue of totality.

It should also be noted that the difference between deterministic (partial) automata and deterministic total automamata is not really of huge importance. One can easily consider a partial automaton as total by adding an extra "error" state. Absent successor states in the partial deterministic setting are then represented by a transition to that particular extra state. The reason why some presentations consider a deterministic automaton to be, at the same time, also "total" or complete is, that, as mentioned, it's not a relevant big difference anyway. Secondly, a complete and deterministic automaton is the more useful representation, either practically or also for other constructions, like *minimizing* a deterministic automaton. But anyway, it's mostly a matter of terminology and perspective: every (non-total) deterministic automaton can immediately alternatively be interpreted as total deterministic function. It's the same in that any partial function from A to B, sometimes written $A \hookrightarrow B$ can be viewed as total function $A \to B_\perp$, where $B_\perp$ represents the set B extended by an extra error element $\perp$.

The automaton from the earlier example, the process scheduler, is *deterministic*.

Runs, acceptance, and languagues of automata

Definition 3.4.4 (Run)**.** A *run* of a finite state automaton $\mathcal{A} = (Q, q_0, \Sigma, F, \to)$ is a (possibly infinite) sequence

$$\sigma = q_0 \xrightarrow{a_0} q_1 \xrightarrow{a_1} \ldots$$

The notation $q \xrightarrow{a} q'$ is meant as $(q, a, q') \in \to$. Each run corresponds to a **state sequence** over Q and a **word** over the alphabet Σ. Of course also the state sequence can be called a word, interpreting the set of states as alphabet.

As mentioned a few times: the terminology is not "standardized" throughout. Here, on the slides, we defined a run of a finite-state automaton as a finite or infinite sequence of *transitions*. Words which more or less means the same in various contexts (an perhaps based on transition systems or similar, not automata) include *execution*, *path*, etc. All of them are modulo details similar in that they are *linear* sequences and refer to the "execution" of an automaton (or machine, or program). The definition we have given contains "full information" insofar that it is a sequence of transitions. It corresponds to the choice of words in [5] (the "Spin-book"). The book Baier and Katoen [1], for example, uses the word run (of a given Büchi-automaton) for an infinite state sequence, starting in a/the initial state of the automaton.

For me, the definition of run as given here is a more "plausible" interpretation of the word. A run or execution (for me) should fix all details that allows to reconstruct or replay what concretely happened. Considering state sequences as run would leave out which *labels* are responsible for that sequence. Not that it perfectly possible that $q \xrightarrow{a} q'$ and $q \xrightarrow{b} q'$ (for two different labels a and b) even if the automaton is deterministic.

In a deterministic automaton, of course, a "word-run" determines a "state-run".

As a not so relevant side remark: we stressed that modulo minor variations, a commonality on different notions of *runs*, *executions*, (and histories, logs, paths, traces . . .) is that they are **linear**, i.e., they are sequences of "things" or "events" that occur when running a program, automaton, ... When later thinking about *branching time logics* (like CTL etc), the behavior of a program is not seen as a set of linear behaviors but rather as a tree. In that picture, one execution correspond to one tree-path starting from the root, so again, one execution is a linear entity.

There exist, however, approaches where **one execution** is not seen as a linear sequence, but as something more complex. Typical would be a *partial* order (a sequence corresponds to a *total* order). There would be different reasons for that. They mainly have to do with modelling concurrent and distributed systems where the total order of things might not be observable. Writing down in an execution that one thing occurs before the other would, in such setting, just impose an artificial ordering, just for the sake of having a linear run, which otherwise is not based on "reality". In that kind setting, one speaks also of partial order semantics or "true concurrency" models (two events not ordered are considered "truely concurrent"). Also in connection with weak memory models, such relaxations are common. Considering partial orders (when it fits) is also a optimization technique: by avoding to explore all interleavings of all linearizations of a partial order, one can make model checking technique more efficient.

Those considerations will not play a role in the lecture: runs etc. are *linear* for us (total orderings).

Example 3.4.5 (Run). Consider the automaton from Figure 3.3. State sequences arising from possible runs look like

$$\mathsf{idle}\ \mathsf{ready}\ (\mathsf{execute}\ \mathsf{waiting})^* \ .$$

Words in Σ that correspond to the shown state sequences are of the form

$$start\ run(block\ unblock)^* \ .$$

There are of course others as well, for instance runs involving the ready-state resp. *preempt*-steps. There are also infinite runs. In general, a single state sequence may A single state may correspond to more than one word, but in the example, the transition-label sequence determines the state sequence: the automaton is deterministic. □

Definition 3.4.6 (Acceptance)**.** An *accepting* run of a finite state automaton $\mathcal{A} = (Q, q_0, \Sigma, F, \rightarrow)$ is a finite run $\sigma = q_0 \overset{a_0}{\rightarrow} q_1 \overset{a_1}{\rightarrow} \ldots \overset{a_{n-1}}{\rightarrow} q_n$, with $q_n \in F$.

In the scheduler example from before: a *state sequence* corresponding to an acceping run is

$$\mathsf{idle}\ \mathsf{ready}\ \mathsf{executing}\ \mathsf{waiting}\ \mathsf{executing}\ \mathsf{end} \ .$$

The corresponding *word* of labels is

$$start\ run\ block\ unblock\ stop \ .$$

A accepting run (as defined here) determines both the state-sequence as well as the label-sequence. In general, the state-sequence in isolation does not determine the label-sequence, not even for deterministic automata. But in the case of the scheduler example, it does. The definition of acceptance is "traditional" as it is based on 1) the existance of an accepting sequence of steps which is 2) finite. The definition speaks of *accepting runs*. With that definition in the background, it's also obvious what it means that an automaton a *word* over Σ or what it means to accept a state sequence. Later, when we come to LTL model checking and Büchi-automata, the second assumption, that of finite-ness will be dropped, resp. we consider *only* infinite sequences. The other ingredient, the $\exists$-flavor (there exists an accepting run) will remain.

Angelic vs. daemonic choice The $\exists$ in the definition of acceptance is related to a point of discussion that came up in the lecture earlier (in a slightly different context), namely about the nature of "or". I think it was in connection with regular expressions. Anyway, in a logical context (like in regular expressions or in LTL), the interpretation is more or less clear. If one takes the logic as describing behavior (the set of accepted words, the set of paths etc.), then disjunction corresponds to *union* of models.

When we come to "disjunction" or choice when describing an automaton or accepting machine, then one has to think more carefully. The question of "choice" pops up only for *non-deterministic* automata, i.e., in a situation where $q_0 \xrightarrow{a} q_1$ and $q_0 \xrightarrow{a} q_2$ (where $q_1 \neq q_2$). Such situations are connected to *disjunctions*, obviously. The above situation would occur where q_0 is supposed to accept a language described by $a\varphi_1 \vee a\varphi_2$. In the formula, φ_1 describes the language accepted by q_1 and φ_2 the one for q_2. The disjunction $\vee$ is an operator from LTL; if considering regular expressions instead, the notations "$|$" or "$+$" are more commonly used, but they represent disjunction nonetheless. *Declaratively*, disjunction may be clear, but when thinking operationally, the automaton in state q_0 when encountering a, must make a "choice", going to q_1 or to q_2, and continue accepting. The definition of acceptance is based on the *existance* of an accepting run. Therefore, the accepting automaton must make the choice in such a way that leads to an accepting state (for words that turn out to be accepted). Such kind of making choices are called *angelic*, the choice supports acceptance in a best possible way. Of course, they are also "prophetic" in that choosing correctly requires foresight (but angels can do that... Daemons can do that as well, it's only that angels make decisions in favor of acceptance whereas daemons use their forsight to sabotage it). Of course, concretely, a machine would either have to do *backtracking* in case a decision turns out to be wrong. Alternatively one could turn the non-deterministic automaton to a deterministic one, where there are no choices to made (angelic or otherwise). It corresponds in a way a precomputation of all possible outcomes and exploring them at run-time all at the same time (in which case one does not need to do backtracking). A word of warning though: Büchi automata may not be made deterministic. Furthermore, it's not clear what to make out of *backtracking* when facing *infinite* runs.

The angelic choice this proceeds successfully if *there exists* a successor state that allows succesful further progress. There is also the *dual interpretation* of a choice situation which is known as *demonic*, which corresponds to a $\forall$-quantification. The duality between those two forms of non-determism shows up in connection with *branching time* logic (not so much in LTL). Also the duality is visible in "open systems", i.e., where one distinguishes the systems from its environment. For instance for security, the envionment is often called *attacker* or *oppenent*. This distinction is at the core also of game-theoretic accounts, where one distinguishes between "player" (the part of the system under control) and the "oppenent" (= the attacker), the one that is not under control (and which is assumed to do bad things like attack the system or prevent the player from winning by winning himself). In that context, the system can try do a good choice, angelically ($\exists$) picking a next step or move, such that the outcome is favorable, no matter what the attacker does, i.e., no matter how bad the demonic choice of the opponent is ($\forall$).

Let's just define what it means that an automaton accepts a word. That also defined the *language* of an automaton as the set of all accepted words.

Definition 3.4.7 (Language)**.** The *language* $\mathcal{L}(\mathcal{A})$ of automaton $\mathcal{A} = (Q, q_0, \Sigma, F, \rightarrow)$ is the set of words over Σ that correspond to the set of all the accepting runs of $\mathcal{A}$.

In general, there are infinitely many words in such languages. Languages of finite-state automata and with this acceptance condition correspond to regular languages, languages expressable via regular expressions.

For the given scheduler automaton from before, one can capture its language of finite words by (for instance) the following regular expression

$$\mathit{start}\ \mathit{run}\ ((\mathit{preempt}\ \mathit{run})^* \ \mid\ (\mathit{block}\ \mathit{unblock})^*)\ \mathit{stop}\ .$$

We use | for "or", we could also have used +.

In the context of language theory, *words* are finite sequences of letters from an alphabet Σ, i.e., a word is an element from Σ^*, and languagues are sets of words, i.e., subsets of Σ^*. As stressed, for LTL and related formalisms, we are concerned with *infinite words* and languages over infinite words.

Reasoning about runs

We are of course mainly interested in LTL as temporal logic in this chapter. But as a warm-up, we can use regular expressions to specify temporal properties.

Let's have a look a the following temporal property

> If first p becomes true and afterwards q becomes true, then afterwards, r can no longer become true

In the context, the desired property is also called a *correctness claim.* As we know from the big-picture discussion about *refutation*, the model checking procedure for LTL works with the *negation* of the specification. In our example we can formulate that as

> It's an **error** if in a run, one sees first p, then q, and then r.

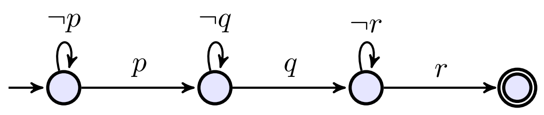


Figure 3.4: Automaton for the negated specification

That property can be represented by the automaton from Figure 3.4.

The accepting states captures that the error condition has been reached. So reaching the accepting state means one has detected about a **violation** of the correctness property.

The example illustrates one core ingredient to the automata-based approach to model checking. One is given a property one wants to verify, like the informally given one from above. In order to do so, one operates with its *negation.* In the example, that negation can be straightforwardly represented as *standard* acceptance in an FSA. Being represented by conventional automata acceptance, the detected errors are witnessed by *finite words* corresponding to finite executions of a system.

Generally, one cannot expect to find a violation of the specification in a finite sequence. A property (like the one above) whose **violation** can be detected by a *finite* path is called a **safety property**. Safety properties form an important class of properties. Note: safety properties are *not* those that can be *verified* via a finite trace, the definition refers to the negation or violation of the property: a safety property can be *refuted* by the existance of a finite run.

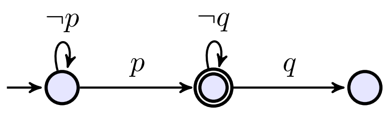
That fits to the standard informal explanation of the concept, stipulating: “that never something bad happens” (because if some bad thing happens, it means that one can detect it in a finite amount of time). The slogan is attributed to Lamport [6]. That “bad” in the sentence refers to the *negation* of the original property one wishes to establised (which is seen thus as “good”). Note one more time: the *original* desired property is the *safety property*, not its negation.

Still another angle to seeing it is: a safety property on paths is a property has the following (meta-)property: If the safety property holds *for all finite behavior, then it holds for all behavior* (all behavior includes *inifinite* behavior). For the mathematically inclined: this is a formulation connected to a *limit* construction or closure or a *continuity* constraint, when worked out in more detail (like: infinite traces are the limit of the finite ones etc).

How to use automato to reason about infinite run?

Let’s also have a look at a livenes property, say “if p then eventually q.”, respectively its negation

It’s an **error** if one sees p and afterwards never q (i.e., forever $\neg q$).



A violation of that is possible only in an **infinite** run. Consequently it cannot be expressed by the *conventional* notion of acceptance.

A moment's thought should get the "silly" argument out of the way that says: "oh, if checking the negation via an automaton does not work easily in a conventional manner, why not use the original, non-negated property. One can formulate that without referring to infinite runs and with standard acceptance.".

Ok, that's indeed silly in the bigger picture of things (why?). What we need (in the above example) to capture the negation of the formula is to express that, after p, there is *forever* $\neg q$, which means for the sketched automaton, that the loop is taken forever, resp. that the automaton stays infinitely long in the middle state (which is marked as "accepting"). What we need, to be able to accepting infinite words is a *reinterpretation* of the notion of acceptance. To be accepting is not just a "one-shot" thing, namely reaching some accepting state. It needs to be generalized to involve a notion of visting states *infinitely* often.

In the above example, it would seem that acceptance could be "stay forever in that accepting state in the middle". That indeed would capture the desired negated property. The definition of "infinite acceptance" is a bit more general than that ("staying forever in an accepting state"), it will be based on "visiting an accepting state infinitely often, but it's ok to leave it in between". That will lead to the original notion of **Büchi acceptance**, which is one flavor of formalizing "infinite acceptance" and thereby capturing infinite word languages.

There are alternatives to that particular definition of acceptance. In the lecture we will encounter a slight variation called *generalized Büchi acceptance*. It's a minor variation, which does not change the power of the mechanism, i.e., generalized or non-generalized Büchi acceptance does not really matter. However, the GBAs are more convenient when translating LTL to a Büchi-automaton format. It may be (very roughly) compared with regular languages and standard FSAs. For translating regular expressions to FSAs, one uses a variation of FSAs with so-called ϵ-transitions (silent transitions), simply because the construction is more straightforward (compositional). Generalizing Büchi automata to GBAs does not involve ϵ-transitions but the spirit is the same: use a slight variation of the automaton format for which the translation works more straightforwardly.

3.4.2 Büchi automata

As mention, Büchi-automata are defined as finite state automata from Definition 3.4.1. What makes them "Büchi" is the different acceptance condition, that can handle infinite words or runs. Infinite runs are often often called ω-*runs* ("omega runs"), and the corresponding acceptance conditio consequently ω-acceptance. The are different versions of the idea (Büchi, Muller, Rabin, Streett, parity etc.,), but here we present **Büchi acceptance** [3] [2].

Definition 3.4.8 (Büchi acceptance)**.** An *accepting* ω*-run* of the finite state automaton $\mathcal{A} = (Q, q_0, \Sigma, F, \rightarrow)$ is an infinite run σ such that some $q_i \in F$ occurs infinitely often in σ.

Automata with this acceptance condition are called **Büchi automata**.

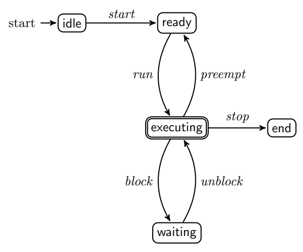


Figure 3.5: Scheduler (2)

Example 3.4.9 (Infinite runs). Consider the automaton from Figure 3.5. It almost the same than the one from Figure 3.3 earlier except that the accepting state has been changed.

One accepted infinite state sequence is the following:

$$\mathsf{idle}\ (\mathsf{ready}\ \mathsf{executing})^{\omega}$$

A corresponding ω-word is:

$$start\ (run\ preempt)^{\omega}$$

□

The automaton is meant to illustrate the notion of Büchi-acceptance; it's not directly meant as some specific logical property (or a negation thereof). Nor do we typically think that transition systems that present the "program" we like to model check work as language acceptors and thus have specific accepting states they have to visit infinitely often. Program run, but the Büchi-automata that results from the translation of an LTL formula do have accepting states meant to check the (negation of the) property of interest.

The symbol ω often stands for "infinity" (here and elsewhere). Actually ω in general stands specific infinity as one can have different forms and levels of infinities. Those mathematical fine-points may not matter much for us. But it's the "smallest infinity larger than all the natural numbers", which makes it an *ordinal number* in math-speak and being defined as the "smallest" number larger than $\mathbb{N}$ makes this a *limit* or *fixpoint* definition. It's connected to the earlier, perhaps cryptic, side remark about safety and liveness, where it's important that infinite traces are the *limit* of the finite ones).

For instance, $(ab)^*$ stands for finite alternating sequences of a's and b's, including the empty word ϵ, starting with an a and ending in a b. The notation $(ab)^{\omega}$ stands for *one* infinite word of alternating a's and b's, starting with an a (and not ending at all, of course). Given an alphabet Σ, Σ^{ω} represents all infinite words over Σ. As a side remark: for non-trivial Σ (i.e., with more than 2 letters), the set Σ^{ω} is no longer enumerable (it's a consequence of the simple fact that its cardinality is larger then the cardinality of the natural numbers).

Sometimes, one finds the notation Σ^∞ (or $(ab)^\infty \dots$) to describe infinite *and* finite words. Remember in that context, that the semantics of LTL formulas is defined over *infinite* sequences (paths), only.

As mentionned shortly, there is also a variation of Büchi-acceptance, called *generalized* Büchi-acceptance. It's a minor variation of the original definition. Both flavors of acceptance conditions are of equivalent expressiveness. When translating LTL-formulas to an infinite-word automaton, the generalized Büchi-automaton format is more convenient, so we will use that one. If one prefers plain Büchi-automata, one could, in a second stage, translate the generalized version into a non-generalized one. But we don't look into that.

For a σ for of a Büchi-automaton, let's write $\mathit{inf}(\sigma)$ for the set of states that occur infinitely often in σ. With this, we can define the generalized Büchi automata and their acceptance condition as follows:

Definition 3.4.10 (Generalized Büchi automaton)**.** A *generalized Büchi automaton* is an automaton $\mathcal{A} = (Q, q_0, \Sigma, F, \rightarrow)$, where $F \subseteq 2^Q$.

Let $F = \{f_1, \dots, f_n\}$ and $f_i \subseteq Q$. A run σ of $\mathcal{A}$ is *accepting* if

$$\text{for each } f_i \in F,\ \mathit{inf}(\sigma) \cap f_i \neq \emptyset.$$

Not that not only the acceptance has changed, but also the format of the format of the automaton. Büchi automata, as standard finite state automata, the acceptance is based on a set F of states. Here, F is not a set of states, F is a set of set of states. So the automata has not multiple accepting states but multiple accepting sets of states.

As mentioned earlier, the motivation to introduce this (minor) variation of what it means for an automaton to accept infinite words comes from the fact that it is just easier to translate LTL into this format.

Büchi automata (generalized or not) is just one example of automata for infinite words. Those are generally known as ω-automata. There are other acceptance conditions (Rabin, Streett, Muller, parity ...), which we will probably not cover in the lecture. When allowing non-determinism, they are all equally expressive. It's well-known that for finite-word automata, non-determinism does not add power, resp. that determinism is not a restriction for FSAs. The issue of non-determinism vs. determinism gets more tricky for ω-words. Especially for Büchi-automata: deterministic BAs are strictly less expressive than their non-deterministic variant! For other kinds of automata (Muller, Rabin, Street, parity), their deterministic and non-deterministic versions are equally expressive. In some way, Büchi-automata are thereby not really well-behaved, the other automata are nicer in that way. The class of languages accepted by those automomata is also known as ω-regular languages.

3.4.3 Stuttering

Next we address an issue not so much from automata theory, but from the use we make of the concepts modelling systems and model checking them, In the big-picture dicussion earlier we mentioned that the approach works with two rather similar representations with nodes and edges and label, namely automata and transition systems. There are pretty close, but they also serve different purposes. The transition systems' role is to model the execution of programs or system, the automata on the other hand represent an LTL formula, and the latter has accepting states, whereas "acceptance" is a concept alien to programs. Instead program may *terminate*.

So that's another mismatch, LTL works on *infinite runs*, and Büchi acceptance, genealized or not, accepts by definition *only* infinite runs.

But that's an easy nut to crack. We cannot allow the system to just terminate, because we can only logically (resp. by Büchi-automata) handle infinite run. So if our transition system (massaged into another Büchi-automaton) terminate, we simply let it artificially continue infinitely by doing nothing. That is known as **stuttering**. This allows to treat finite and infinite acceptance uniformely by Büchi-acceptance condition. This avoids coming up with an more complex alternative acceptance condition that allows to accept finite and infinite words.

Let ε be a predefined nil symbol and the alphabet/label set extended to $\Sigma + \{\varepsilon\}$. So a finite run terminating by reaching some state without successor becomes, by stuttering, an inifite ones, where, at some point, infinitely often ε is done. Stuttering is allowed only at the end, i.e. a run must end in an end-state

Definition 3.4.11 (Stutter extension)**.** The *stutter extension* of a finite run σ with last state s_n, is the ω-run

$$\sigma\ (s_n, \varepsilon, s_n)^\omega\ . \tag{3.18}$$

Let's revisit the schedular example again

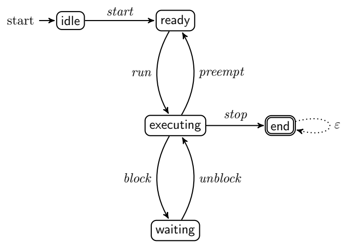


Figure 3.6: Scheduler with stuttering

Example 3.4.12 (Stuttering). The "process scheduler" example from Figure 3.3 uses the accepting state end now as natural end state with an ε-loop which is also accepting (see Figure 3.6). Examples of accepting state sequences resp. accepting words corresponding to an accepting ω-run include the following:

$$\text{idle ready executing waiting executing end}^\omega$$

and

$$\textit{start run block unblock stop}^{\omega}$$

□

So far, we have introduced the stutter extension of a (fintite) run. But runs will be ultimately runs “through a system” or through an automaton. Of course there could be a state in the automaton when it’s “stuck”. Note that we use automata or transition systems to represent the behavior of the system we model as well as properties we like to check. The stutter-extension on runs is concerned with the “model automaton” representing the system. To me able to judge whether a run generated by the system satisfies an LTL property, it needs to be an *infinite run*, because that’s how $\models$ for LTL properties is defined. The fact that in the construction of the algorithm, also the LTL formula (resp. its negation) will be translated to an automaton is not so relevant for the stutter discussion here.

3.4.4 Something on logic and automata

In this section we bridge a mismatch between transition systems and Büchi automata. The mismatch, mentioned earlier is that automata are edge labelled and transition systems, at least the ones we introduced, carry information in the worlds or states. The mismatch is not large, so bridging it will be easy

Another issue we look at is how to see at different Büchi automata can represent LTL properties. Also that is done informally, we don’t show the actual construction. That is for later.

From Kripke structures to Büchi automata

We have encountered different “transition-system formalisms”. One under the name transition systems (or Kripke models or Kripke structures), the other one automata. [1] talk about transition systems instead of Kripke structures (and they allow labels on the transitions, as well).

The Kripke structures or transition systems are there to model “the system” whereas the automata serve to specify temporal properties of the system.

On the one hand, those formalisms are basically the same. On the other hand, there is a slight mismatch: the automaton is seen as “transition- or edge-labelled”, the transition system is “state- or world-labelled”. The mentioned fact that the transition systems used in [1] are additionally “transition-labelled” is irrelevant for the discussion here, the labelling there serves mostly to be able to capture synchronization (as mechanism for programming or describing concurrent systems) in the parallel composition of transition systems.

As also mentioned earlier, there is additionally slight ambiguity wrt. terminology. For instance, we speak of states of an automaton or a state (= world) in a transition system or Kripke structure. On the other hand, we also encountered the state-terminology as a mapping from (for example propositional) variables to (boolean) values. Similar ambiguity is there for the notion of *paths*. It should be clear from the context what is what. Also the notions are not contradictory. We will see that for the notion of "state" later, as well.

Now, there may be different ways to deal with the slight mismatch of state-labelled transition system and edge-labelled automata on the other. The way we are following here is as follows. The starting point, even before we come to the question of Büchi-automata, describes the behavior of Kripke structures in terms of statifaction per *state* or "world", not in terms of *edges*. For instance $\Box\Diamond p$ is true for a path which contains infinitely many occurrence of p being true, resp. for a Kripke structure whose every run corresponds to that condition. So, for all infinite behavior of the structure, p has to hold in infinitely many *states* (not transitions); propositions in Kripke-structures hold in states, after all (or Kripke structure are state labelled).

Remember also that we want to to check that the "language" of the system M is a subset of the language described by a LTL-specification φ, like $M \models \varphi$ corresponds to $[\![M]\!] \subseteq [\![\varphi]\!]$. To do that, we'd like to translate LTL-formulas (more specifically $\neg\varphi$) into automata, but those are transition-labelled (as is standard for automata in general). So, $[[M]] $ is a language of infinite words corresponding to sequences of "states" and the state-attached information. On the other hand, $[\![\varphi]\!]$ is a language containing words referring to edge-labels of and automaton.

So there is a slight mismatch. It's not a real problem, one could easily make a tailor-made construction that connects the state-labelled transition systems with the edge-labelled automaton and then define what it means that the combination is does an accepting run. And actually, in effect, that's what we are doing in principle. Nonetheless, it's maybe more pleasing to connect two "equal" formalisms. To do that, we don't go the direct way as sketched. We simply say how to interpret the state-labelled transition system as edge-labelled automaton, resp. we show how in a first step, the transition system can be transformed into an equivalent automaton (which is straighforward). Thus we have two automata, and then we can define the intersection (or product) of two entities of the same kind.

One might also do the "opposite", like translating the automaton into a Kripke-structure, if one wants both logical description and system desciption on equal footing. However, the route we follow is the standard one. It's a minor point anyway and on some level, the details don't matter. On some other level, they do. In particular, if one concretely translates or represents the formula and the system in a model checking tool, one has to be clear about what is what, and which representation is actually done.

Translating transition systems to Büchi automata

We call the "states" here now W for worlds, to distinguish it from the states of the automaton. We write $\rightarrow_M$ the accessibility relation in M, to distinguish it from the labelled transitions in the automaton.

Definition 3.4.13 (Translating a Kripke structure into an Büchi-automaton)**.** Given $M = (W, R, W_0, V)$. An automaton $\mathcal{A} = (Q, q_0, \Sigma, F, \rightarrow)$ can be obtained from M as follows. The alphabet for the transition labels is given as $\Sigma = 2^P$. For the states, set

$$Q = W + \{i\} \qquad q_0 = i \qquad F = W + \{i\}$$

For the *transitions*, we set $s \xrightarrow{a} s'$ iff $s \rightarrow_M s'$ and $a = V(s')$ $s, s' \in W$ and $i \xrightarrow{a} s \in T$ iff $s \in W_0$ and $a = V(s)$.

Note: the translation turns *all* worlds of the transition system into accepting states of the Büchi automaton. Basically, the accepting conditions are not so "interesting" and making all states accepting means: I am interesting in *all* behavior as long as it's inifinite. The KS (and thus the corresponding BA) is not there to "accept" or reject words. It's there to produce infinite runs without stopping (and in case of an end-state it means, conntinue infinitely anyway by *stuttering*).

Here, the Kripke structure has initial states or initial worlds (W_0), something that we did not have when introducing the concept in the modal-logic section. At that point back then we were more interested in questions of "validity" and "what kind of Kripke-frames is captured by what kind of axioms", things there are important in dealing with validity etc. In that context, one has no big need in particular "initial states" (since being valid means for all states/worlds anyway). But in the context of model checking and describing systems, it's, of course, important.

Note also, that the valuations $V : W \rightarrow (P \rightarrow \mathbb{B})$ attach to each world or "state" a mapping, that assigns to each atomic proposition from P a truth value from $\mathbb{B}$. That can be equivalently seen as attaching to each world a *set* of atomic propositions, i.e., it can be seen as of type $W \rightarrow 2^P$. Perhaps confusingling the assignment of (here Boolean values) to atomic propositions, i.e., functions of type $P \rightarrow \mathbb{B}$, are sometimes also called *state* (more generally: a state is an association of variables to their (current) value, i.e., a state is a current content or snapshot of the memory). The views are not incompatible (think of the program counter as a variable ...)

Example 3.4.14 (From Kripke structure to Büchi automata). A Kripke structure (whose only infinite run satisfies (for instance) $\Box q$ and $\Box\Diamond p$):

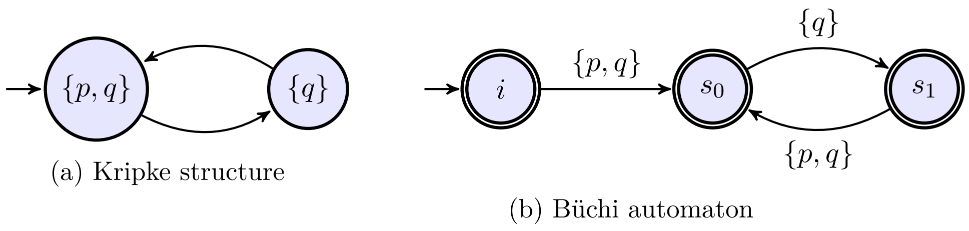


Figure 3.7: Translation

□

3.4.5 Describing temporal properties by LTL and Büchi automata

We have two formalisms to describe sets of infinite paths: LTL and Büchi-automata. Even three, if we count transition systems in. As mentioned in the big-picture discussion, the situation may be compared to standard word languages, with words of finite length, which can be described by regular expressions and finite state automata. Finite state automata and regular expressions are equivalent in that they can describe the same word languages.

For LTL and Büchi-automata, they are *not* two versions of the same thing, only half so: for every LTL formula φ, there exists a Büchi automaton that accepts precisely those runs that satisfy φ. The reverse direction does not hold, i.e., Büchi-automata are more expressive.

Here we illustrate how one can use Büchi automata to capture some LTL properties, without showing the contruction; that comes later. Here, we also mention some other formalisms for infinite languages, some more expressive than LTL, some less, some incomparable.

Example 3.4.15 (Stabilization: "eventually always p"). Figure 3.8 shows an Büchi-automaton checking the LTL property $\Diamond\Box p$. □

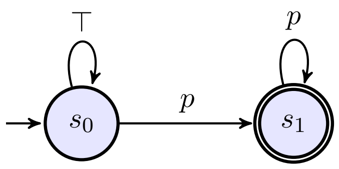


Figure 3.8: Büchi-automaton for stabilization

(Lack of?) expressiveness of LTL

As mentioned, the analogy of Büuchi-automata and LTL on the one and and FSa and regular expressions on the others is not 100%. In the case of finite-word languages, the two mechanisms of automata and of regular expressions are equivalent. Here, LTL is strictly **weaker** than BA.

The Büchi-automata of the following example captures a property that cannot be expressed LTL.

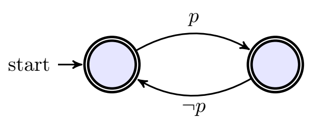
Example 3.4.16. The example is about capturing the following temporal property:

p is always false after an *odd* number of steps

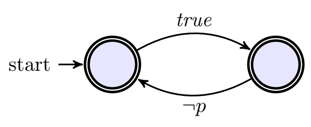
Let's start by trying to capture that with LTL:

$$p \land \Box(p \rightarrow \bigcirc\neg p) \land \Box(\neg p \rightarrow \bigcirc p) \tag{3.19}$$

The formula from equation (3.19) indeed assures that p is false after odd steps, as required, but it's not exactly what was informally given! The LTL specifies, with p holding initially, a strict oscillation between places where p holds and those where p does not hold. But that's more than what the informal sentence required, which did not insist that on the even places, p holds. At any rate, this oscillating behavion can be represented by the Büchi-automaton from Figure 3.9a.



(a) Strict oszillation



(b) $\neg p$ after odd steps

Figure 3.9: Büchi automata

The second Büchi-automaton from Figure 3.9b does not impose restriction on the even-positions and thus corresponds to the original informal formulation. LTL cannot capture that automaton, though we don't prove it.

The property can be captured by the following temporal formula:

$$\exists t.\ t \land \Box(t \rightarrow \bigcirc\neg t) \land \Box(\neg t \rightarrow \bigcirc t) \land \Box(\neg t \rightarrow p) \tag{3.20}$$

The formula, however, is not LTL. It's a formula from what is called ∃LTL. □

What do do about the mismatch? One can live with the mismatch, of course. That's what we do in the lecture: we translate LTL Büchi-automata for the purpose of model checking, and that's all we are really want.

But both formalisms seems kind of natural, so one can try to repair the mismatch (or at least analyze the reason of the mismatch).

Concerning Büchi-automata, there exist also so called ω-**regular** expressions and ω-regular languages, which are a generalization of regular languages and whose expressiveness matches that of non-deterministic Büchi-automata. They look like regular languages, except one can write r^ω (not just r^*). There exist a "crippled" form of "infinite regular expressions" that is an exact match for LTL.

To "repair" the mismatch between ω-regular languages on the one hand and LTL on the other, one could two things. One can ask, what needs to be taken away from BAs resp. ω-regular expressions to make them fit to LTL, resp. ask whethe there an automaton model that exactly fits LTL? Alternatively one can ask: what needs to be added to LTL to make it as expressive as BAs? Example 3.4.16, in particular the formula from equation (3.20) hints at a solution for the second approach: It's a result from [4] which states that allowing prefix *existential quantification* over one propositional variable (the t in the example) is enough to repair the mismatch. That version if LTL is sometimes called "existential LTL" or ∃LTL.

The Spin model checker resp. its input language Promela offers another mechanism that gives the same expressivity to LTL as ω-regular languages. This mechanism is known as **never claims**.

Spin has a translator `ltl2ba` and one can find translators from LTL to BA on the net as well, for instance under `http://www.lsv.fr/~gastin/ltl2ba/`.

Figure 3.10 compared different well-known temporal logics wrt. their expressivity. Concerning LTL, let's remark that when removing the next-operator ¬, the logic becomes strictly less expressive. That fragment is of interest when specifying and verifying properties of *asynchronous* systems. By that we mean systems concsisting of a processes or threads running concurrently, where the next step of the system seen globally is in general done by one of the processes, chosen randomly, and the other processes don't do anything. Practically, the choices is done perhaps by a scheduler, that picks among the enabled processes on to execute. Typically the scheduler chooses not really randomly, not does it roll the dice freshly after every indidual step of a process. Normally some pricorities or stategy is involved (round-robin, maybe), and every process gets time-slots to do quite a number of steps before pre-empted again. But for modelling, one may chose to abstracting away from the scheduler, and treating it as doing random scheduling. That has not just the advantage of simplifying the model and this the model-checking challenge. Additionally, if one has model-checked a concurrent program as ok under random-scheduling, it works for any specific strategy a real scheduler may apply.

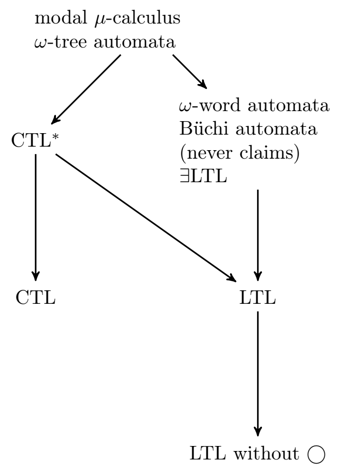


Figure 3.10: Expressivity of a few temporal logics

Now, in such a asynchronpus picture, and if $\bigcirc$ speaks about transitions of the global system and if on specifies local properties of one process, then $\bigcirc\varphi$ make little sense. For instance, if one process is in a state doing `x:=1`, it still makes no sense to say locally "$\bigcirc(x = 1)$". In the face of non-deterministic, asynchronous scheduling, it's not clear what the next global transition will be, and it makes not much to specify what should hold. One can still do it, but it would require global knowledge, and the argumnt here is specifying locally, which is less messy. Locally one could say $\Diamond(x = 1)$ instead of $\Diamond(x = 1)$, assuming that the scheduler is *fair* (but otherwise free to choose randomly).

3.4.6 Automata products

In the big-picture discussion about model-checking as refutation, one of the step is to calculate the intersection

$$[\![M]\!] \cap [\![\mathcal{A}_{\neg\varphi}]\!] \tag{3.21}$$

The interection or conjuction construction is done on the corresponding automata. So, given two automata, we need to construct an automaton that represents to the intersection of the corresponding languages (or the conjuction of the corresponding properties).

In principle, such constructions are known from standard automata. It also called the (or a) **product** construction as the states of the joint automaton consists of pairs of states from the contibuting automata.

The constructiion for Büchi-automata is more complex than for standard FSAs, and the complication concerns, not suprisingly, the accepting states. For standard automata, and accepting state of the product automaton is simply tuples of the original automata: a finite word in the intersection is accepted if both component automata reach one of their respective accepting state, it's very straightforward. For Büchi-automata, it's no longer that easy.

Indeed, in this section we **won't** present the general product construction for Büchi-automaton, because we don't need it in its full generality. We are after intersection in the situation of equation (3.21). In particular, the M is the Büchi-automaton that corresponds to the transition system under investigation. What makes it special is that, as Büchi automaton, all its states are accepting (and never gets really stuck since it's stuttering). In this situation, the only contributing factor to the acctance of the product is the Büchi automaton from the formula. That's as it should be anyway, the specification determines whether property φ holds or not, the system M does what it does, and the accepting states of the product are solely determined by the formula.

Two kinds of products

In fact, the section discusses two kinds of automata constructions, i.e., composition operators on automata, called *synchronous* and *asynchronous* product. Both serve two different purposes. The synchronous product is the one we just sketched and will capture language *intersection*. It corresponds to the standard product construction known from standard FSAs. As also said, we only cover the product or intersection between two Büchi-automata, where one is special insofar that it comes from turning the program into such an automaton. In that sense, the way the synchronous product used here is *asymmetric* (normally products are symmtric, i.e., commutative).

The *asynchnonous* product here comes from the underlying compitation model. The presentation here is based on [5] and thus influenced by the choices as made in the Spin model checker. In Spin, systems consists typically of a number of processes running in parallel or concurrently. The input language of the Spin model checker, called Promela, resembles C. Processes run concurrently, communicate via shared variables and via channels. The semantics is a typical **interleaving** semantics. If we ignore synchroniation by channels, processes do their steps independently, i.e., it's an **asynchronous** execution model.

That kind of behavior is called **asynchronous** product here in the lecture (where we consider the system processes as automata). As said, when thinking in terms of processes or threads, one would not use the word "asynchronous product" but rather say, the processes run in parallel or concurrently and that in an indepedent or **asynchonous** manner (though synchronization when needed can be achieve by channels or locks. . .).

There are other forms of concurrency, for instance for systems with a global clock, where there processes run in lock-step. In that case, the processes run 'synchronlously* in parallel (with the clock as global synchronizing mechanism). And in such a setting, parallel composition reps. the product would be *synchronous*

In our setting here, processes run under an interleaving model ("asynchronously") and the composition of the system with the "formula" is done synchronously. The automaton representing the formula is there to observe or monitor the system, and this needs to be done fully in-sync with the model.

While talking about processes in Spin/Promela: as said, they don't run under a global clock as synchronizing mechanism. Instead, they can synchronize and communciate via *channels*. In our presentation, channels won't play a role. Nonetheless, automata equipped with buffered channel communication between them are a well-established model for *protocol specification*. Spin's programming or modelling language is basically processes + channels. That's a popular foundation for protocol verification. Formal models in that direction are known under various names and different notations. One name is *extended finite state automata* where extended means "extended by communication buffers" (mostly FIFO buffers).

Asychronous product

Let's start with the *asynchnonous* product, here of two automata. In the product, a step of the combined automaton consist of a step of one automaton where there other one does nothing

Definition 3.4.17 (Asynchronous product)**.** The *asynchronous product* of two automata A_1 and A_1, (written $A_1 \times A_2$, or $A_1 \parallel A_2$) is given as $(Q, q_0, \Sigma, F, \rightarrow)$ where

- $Q = Q_1 \times Q_2$,
- $q_0 = q_0^1 \times q_0^2$,
- $\Sigma = \Sigma_1 \cup \Sigma_2$, and
- $F = \{(q_1, q_2) \mid q_1 \in F_1 \ \text{ or } q_2 \in F_2\}$.

$$\frac{q_1 \rightarrow_1 q_1'}{(q_1, q_2) \rightarrow (q_1', q_2)} \ \textsc{Par}_1 \qquad \frac{q_2 \rightarrow_2 q_2'}{(q_1, q_2) \rightarrow (q_1, q_2')} \ \textsc{Par}_1$$

The product is defined for the *binary* case, i.e., the product of two automata. The definition is symmetric and associative, i.e. $A_1 \times A_2 = A_2 \times A_1$ and $A_1 \times (A_2 \times A_3) = (A_1 \times A_2) \times A_3$. The automata left and right of the equations are equal in the sense if being *isomorphic*. With associativity and commutativity, we can make use of n-ary product, i.e., the product of n automata, for which one can write $\prod A_i$. We could establish that there is also a neutral element wrt. the product, and then the $\prod_{i=1}^{n} A_i$ is also defined for $n = 0$ —the empty product should correspond to the neutral element— but let's not bother).

The core of the above definition is the way the *steps* are defined. The composed automaton can make a step, of either the left or else the right automaton can make a step.

Other ingredients of the definition are more a matter of taste. For instance, we have based the definition on automata with one initial state. One can easily do the "same" product construction in case one has automata with multiple initial states.

Another point is the alphabet: here we take the union of the alphabets. Some presentations would say one can compose only automata over the same alphabet (but also that is a non-central point as one can always "extend" the Σ_1 and Σ_2 to a common alphabet, before doing the composition).

More subtle is the definition of the final states. Remember also that the format of the automaton does not distinguish between standard automata and Büchi automata. The distinction is not based on the "format" or syntax of the automation, it's based in the interpretation of the automaton, in particular the interpretation of the acceptance set.

But for us, we don't care here much: remember that the automata are actually transformed from the system programs or processes. Those don't really have accepting state, resp. after the transformation from transition system or Kripke structure into an automaton, *all* states are accepting (which is a way of saying, that acceptance does not really matter). The good news is: if A_1 and A_2 are of that form that all their states are accepting, then that also holds for $A_1 \times A_2$ in the above definition.

Let's have a look as some toy problem represented implemented as two processes running asynchronously.

Example 3.4.18 ($3n + 1$ problem). Assume 2 non-terminating asynchronous processes or automata A_1 and A_2 and a shared variable x. A_1 tests whether the value of x is odd, in which case updates it to $3 * x + 1$. A_2 tests whether the value of a variable x is even, in which case updates it to $x/2$.

The question is: Does the corresponding function *"terminate"* for all inputs x? Let's formulate "termination" as the following LTL property

$$\Box\Diamond(x \geq 4) \qquad \text{or negated: } \Diamond\Box(x < 4) \tag{3.22}$$

□

In the problem, "termination" is meant reaching the endless cycle $4 \to 2 \to 1 \to 4 \ldots$.

The "$3n + 1$" problem is a long-standing open problem. It's known under different names (Collatz's problem, Hasse-Collatz problem, Ulam's conjecture, Kakutani's problem, Thwaites conjecture, Hasse's algorithm, or the Syracuse problem). The problem is not intended of what model checking can do or is good at. It's not even intended to illustrate a *typical* way asynchronous products are used in practice. Remember that the asynchronous product is intended to model concurrency. The problem, however, is *not* concurrent.

What makes it also atypical for standard model checking is that it's an *infinite* problem. The formulation makes a statement for arbitrary n, and that's intended for infinitely many natural numbers (the problem is not meant as saying the function terminates for all numbers up-to `MAXINT` ...). The dependence on some input or some other parameter makes it a *parametrized* problem and *parametrized model checking* is a sub-genre that tries to come up with techniques dealing with parametrization. The parameter can be an input (like for the $3n + 1$-problem), but more conventional would be to check some property for a system $P_1 \parallel \ldots P_n$ consisting of an arbitrary number n of copies of the same system, where all the P_1 are identical (perhaps up-to their process identity). Think of using model checking mutual exclusion not for an implementation of the the 5-philophers problem but for the n-philosophers problem.

Remarks about the notation in the example and their interpretation The asynchronous product is given actually slightly (but not conceptually) deviating from the mathematical definition given above. The definition given before was for automata, but now, we are considering "processes" or Kripke structures or transition systems. But also that is not 100% correct according to the earlier definition or at least not at first sight. Transition systems were introduced as "graphs" where the states give values to atomic propositions. That's a very low level way of seeing things.

The example make use of transition system which allow more convenience in notation (one could call it "applied transition systems" or "symbolic"). It's still ultimately the same, but we allow to have programming variables (x in this case) which can be changed and checked. We don't have just a set of propositional variables any more, but *predicates* over the programming variables (like $even(x)$ and $odd(x)$). As far as the logic is concerned, that seems to go into the direction of "first-order LTL" (having sorts and predicates and variables), but we are not actually doing that, for instance we don't introduces $\forall$ and $\exists$, which gives first-order logic it's actual power). The extension is more to the transition-system side of things and $odd(x)$ is not so much seen as a predicate of the logic, but a boolean expression or guard as used in the underlying programming languages. On the LTL side of things, $odd(x)$ can be seen as a proposition which is either true or not depending on the current value of x. So the interpretation of the transition systems and their behavior and how it connects to LTL should be fairly transparent.

The transition systems here also "deviate" from the core definition in that the transitions are *labelled*. The labels are **not** primarily meant as being symbols from an alphabet that the LTL speaks about. LTL speaks about the states. The transition labels are here represent actions that help to specify what the ("symbolic") transition system does when taking the edge. The interpretation of the label $x := 3x + 1$ is fairly obvious, the intension is that the value of x is accoringly modified when going from the source-state to the target state. The other kind of transition is marked by a boolean condition or guard ($odd(x)$ or $even(x)$). The intention is that it's a *conditional transition* that can be taken if the guard is *true* in the source state.

Spin, resp. Promela allows that kind of statements. The language called Promela quite resembles C at the surface (one can also refer to native C), but there is one point where the semantics of Spin **deviates significantly** from C. Writing in C the sequence

```
(x%2); x = 3x+1; ...
```

simply calculates the remainder of x modulo 2, then forgets the outcome and updates the value of x according to the expression on the right-hand side of the assignemt. In other words, the above snippet can be simplified to `x = 3x+1`.

In Spin, in contrast, the first expression is interpeted as a so-called **guard**: the expression `x%2` is calculated and the outcome determines whether to continue or not. In the tradition of C, in case the outcome is `0`, it's interpeted as `false`, if different from `0`, it's seen as `true`. That means `x%2` corresponds (in a C-typical formulation) to the predicate or guard $odd(x)$. In case the guard evaluates to true, it does not do anything (as in C), in case it evaluates to false, it **blocks** and prevent the rest of the process from proceeding. So, it acts as a **synchronizing** construct:, the transition is enabled or not depending on the "circumstances". Of course, the circumcstances, i.e., the value of x, can change by interference from a second process, at which point the transition becomes enabled and the process can thereby proceed.

This explanation should be enough to understand the example. A few words on guards in concurrent programs might still be of general interest. Syntactically, the solution in Spin is a bit obscure one may say. It basically says, if one uses an expression in place of a statement, then the expression has *synchronizing powers* (like stopping the execution of the process). That's a truly *radical* change of interpretation of expressions! Synchronization is at the core of concurrent programming; hence it's probably a bad idea to hide some key ingredient, conditional guards, in reinterpreting expressions ("BTW, expressions now mean sometimes something really novel and powerfull compared to C, even if they look the same"). in defense one could say, a decent C programmer would probably not use expressions as assignments anyway, bit still.

Other languages would make the special nature of guards more obvious, namely by introducing *special* syntax for it. If b is a boolean expression, then if one wants to use it with synchronizing power, the programmer would have to write special syntax, writing perhaps `await(b)` or similar, making that transparent.

For people experienced in concurrency programming, there is another concern wrt. to guards like *odd* or similar (not really present in the quite simple example). The previous discussion concentrated on "syntactical" issues or language pragmatic (like questioning the wisdom of avoiding special syntax). The point now may be even more serious. In the transition systems below (and in the code snippet above), there are two steps, namely first the guard is check, for instance $odd(x)$ and, in case the guard had evaluated to true and after taking the guard-labelled transition, then the assignment is done. The problem is: the guard itself has no effect (guards and expression are supposed to be side-effect free); it is intended to enable (or not) the subsequent effect. The problem is: whether or not the guard is true is checked but that may *change* after the *odd* transition and before taking the subsequent $x := 3x + 1$ transition. In other words, the two transitions are *not atomic* which may in general lead to problems. On a related note: it's also questionably whether the assignment $x := 3x + 1$ is guaranteed to be atomic.

To be useful as concurrent programming languare or language, Spin offers the possibility enforce **atomicity**. Sping supports slightly different levels of guaranteeing atomicity, but the details don't concern us here.

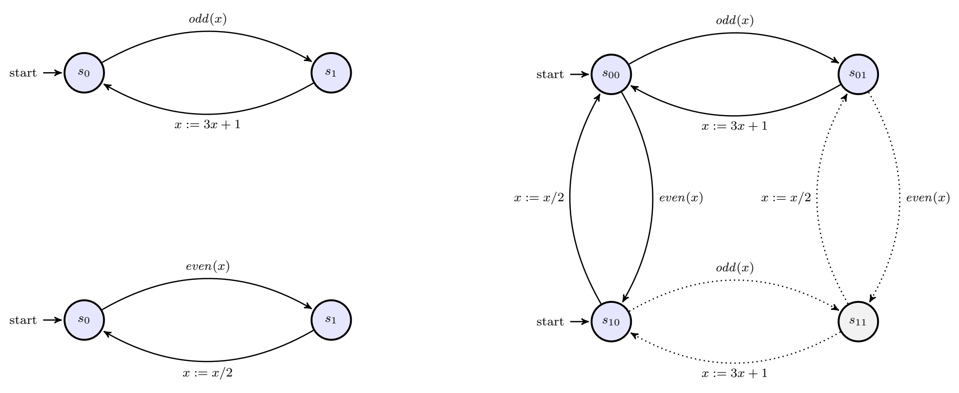
Basically one can *group* the two steps togther, $[odd(x); x := 3x + 1]$ where we use brackets [and] to denote that the execution should be atomic resp., without interleaving. Spin does not use brackents but would use keywords like `atomic{...}` or `d_step{...}`.

In any case: the shown pattern is rather common: in an atomic section, the first one is the guard, and the rest is the effect. It is not so comming (and a bit tricky) to use guards in the middle of an atomic section. Since that pattern is so common, other languages would offer specially syntax for it `await(b){statements})` and there are different names for it (conditional critical region etc). Also related in the notion of *guarded commands*. In transition systems, instead of having two separate transation, one would cound then have labels of the general formal $\overset{g \triangleright a}{\rightarrow}$, where g is the guard and a the action or effect. In our case, one label could be $odd(x) \triangleright x := 3x + 1$.

Anyway, atomicity is not really a problem in our particular (not very typical) example, as the $3n + 1$-problem is not really concurrent anyway.

After all the background information, back to the example and the transition systems for them

Example 3.4.19 ($3n + 1$-problem: asynchronous product). The program informally described in Example 3.4.18 can be represented by the transition systems from Figure 3.11. In the product automaton, the state s_{11} is /unreachable/. When starting at the initial



(a) A_1 and A_2 separately (b) $A_1 \times A_2$

Figure 3.11: $3n + 1$-problem transition systems: asynchronous product

state, the dotted arrows can never be taken. □

Pure automata or transition systems We have made use of the more high-level version of the transition systems, referring to variables like x and transitions with labels that have a "semantics". Besides that, the transition system description as "symbolic", not concrete as it referred to x and not a concrete value of x. So, the description was still a "parametrized problem". We are here *not* looking into parametrized problems. We can only model check *concrete* (non-abstract, non-symbolic) models. So we need to pick a concrete initial value for x. Example 3.4.20 below chooses $x = 4$. This leads to what Holzmann calls a *pure* transition system. Now, the value of x becomes part of the "transition-system state". So the "state" or world now consists of the control-flow state (for instance initially (s_0, s_0) written also s_{00}) together with the state of the memory, i.e., the value of x.

Example 3.4.20 (3n + 1-problem: Pure transitition system). If we start the transition system of the 3n + 1-problem from Example 3.4.19 with a value of 4, we obtain the following concrete or pure transition system from Figure 3.12.

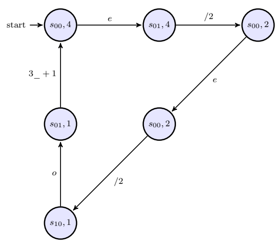


Figure 3.12: Pure automaton: terminal loop

□

In the states, $(s_{00}, 4)$ is supposed to represent a state where both of the original automata A_1 and A_2 are in their respective initial state s_0 and where x has the value 4.

The edge labels can be ignored; they are needed only for the non-pure representation, in the pure transition system they are just kept for readability. In particular: the LTL formula specifying termination does *not* speak about those labels.

We make "short labels" where e and o stands for $even(x)$ and $odd(x)$ in case of the transitions corresponding to guards, and the action labels are similarly shortened. As said, for us, the labels are not really part of the pure transition system anyway. For example, $even(x)$ is true in state (for instance) $(s_{00}, 4)$ anyway (with x understood as carrying the even value 4). So that fact is just captured by a transition $(s_{00}, 4) \to (s_{01}, 4)$ (and the label is more a reminder for the ready why there is that transition). Since, in that state, the guard $odd(x)$ is false, there are no outgoing transitions marked with $odd(x)$: true guards are represented by transitions in the pure representation, false guards represented by the absence of a transition.

Synchronous product

As said, we define the *synchronous product* for 2 Büchi-automata *in a special case*, where for one of the automata, all its states are accepting. That is the case we are interested in here, where one of the BAs, the one describing the system, is translated from the transition system or Kripke structure. In that translation, all states are marked as accepting, so we can focus on that special case.

Indeed, the general case for two arbitrary BAs is slightly more complex. The slightly tricky part would be how to define the accepting states of the product. Apart from that, it's straightforward.

The general intention of such a "product" construction is that he composed machine accepts the *intersection* of the languages of its two constituents. That's conceptually achieved that both automata run in lock-step. For standard (non-Büchi) FSA, a commong word is accepted, if both A_1 and A_2 reach a respective accepting state. That is easy.

For BAs, we have repeated reachability of accepting states, it becomes more tricky: each A_1 and A_2 has to reach at least one of its accepting states infinitely often. But it's not that they re-visit their respecting accepting state always at the same time! That makes the general construction a bit more tricky (but not really challenging; one can figure it out oneself). If one of the automata, say A_1, has *only* accepting states, things get more easy: one can basically ignore A_1, and base acceptance of the product construction in the accepting states of A_2 alone. That's also intuitively what we want to do in model checking. The acceptance condition of A_2 represents a LTL requirement we want to check. The other automaton just "executes", there are no good runs or bad run in A_1 as such, the goodness/badness of the runs is judged by the acceptance conditions in A_2.

It should be noted that in Holzmann [5] the product automaton for NBAs is defined incorrectly (and in previous years, that error showed up also in our slides). As another side remark: for *generalized* BAs, the construction of the accepting conditions for the sychronous product is slightly more elegant compared to standard BAs.

Definition 3.4.21 (Synchonous product (special case))**.** The *synchronous* product of two finite automata $\mathcal{A}_1$ and $\mathcal{A}_2$ (written $\mathcal{A}_1 \otimes \mathcal{A}_2$), for the special case where $F_1 = Q_1$, is defined as finite state automaton $\mathcal{A} = (Q, q_0, \Sigma, F, \rightarrow)$ where:

- $Q = Q_1 \times Q_1$
- $q_0 = (q_{01}, q_{02})$
- $\Sigma = \Sigma_1 \times \Sigma_2$.
- $\rightarrow = \rightarrow_1 \times \rightarrow_2$
- $(q_1, q_2) \in F$ if $q_2 \in F_2$

There is a small final piece we need to take care of, that is stuttering. We discussed the issue already. LTL works on infinite paths, so it must be prevented that that transition system just stops progressing. That's easy to achieve, just let a terminated system continue doing extra do-nothing steps. That is called *stuttering*. The following Definition 3.4.22 adds stuttering to the resulting Büchi-automaton (not the transition system).

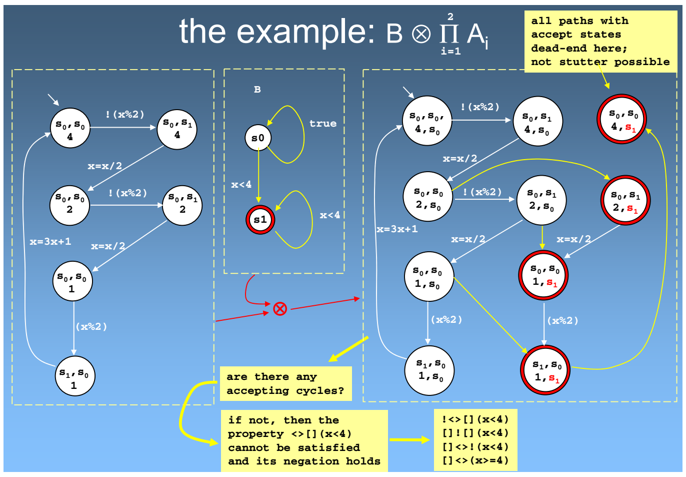
Definition 3.4.22 (Stutter closure)**.** Given a finite-state automaton $\mathcal{A}$ (Büchi or otherwise), the stutter-closure of $\mathcal{A}$ corresponds to $\mathcal{A}$ (same states, labels, same initial and final states) except that some extra ε-labelled self-loops added to the transitions for each state without outgoing transition.

Example: synch. product for $3n + 1$ system and property

Example 3.4.23 ($3n + 1$-problem: product)*.* For the $3n + 1$-example of this section, the product of the automaton from Figure 3.12 with the BA for the LTL property of equation (3.22) from Example 3.4.18 is shown in Figure 3.13.

□

- We require the stutter-closure of P (as P is a finite state automaton (the asynchronous product of the processes automata) and B is a standard Büchi automaton obtained form a LTL formula



all paths with accept states dead-end here; not stutter possible

are there any accepting cycles?

if not, then the property <>[](x<4) cannot be satisfied and its negation holds

```
!<>[](x<4)
[]![](x<4)
[]<>!(x<4)
[]<>(x>=4)
```

Figure 3.13: Product of system automaton and automaton representing the specification

- Not all states necessarily reachable from q_0
- Main difference between asynchronous and synchronous products: labels and transitions. for synchronous product:
  - *joint* transitions of the component automata
  - labels are pairs: the combination of the two labels of the original transitions in the component automata
- In general *here* $P \otimes B \not\equiv B \otimes P$, but given that in SPIN B is particular kind of automaton (labels are state properties, not actions), we have then $P \otimes B \equiv B \otimes P$

3.5 Model checking algorithm

3.5.1 Preliminaries

Algorithmic checking for emptyness

- for FSA: emptyness checking is easy: *reachability*
- For Büchi:
  - more complex acceptence (namely ω-often)
  - simple, one time reachability not enough

$\Rightarrow$ "repeated" reachability

$\Rightarrow$ from initial state, reach an accepting state, and then again, and then again ...

- cf. "lasso" picture
- technically done with the help of SCCs.

Strongly-connected components

Definition 3.5.1 (SCC)**.** A subset $S' \subseteq S$ in a directed graph is *strongly connected* if there is a path between any pair of nodes in S', passing only through nodes in S'. A *strongly-connected component* (SCC) is a *maximal* set of such nodes, i.e. it is not possible to add any node to that set and still maintain strong connectivity.

maximality.

SCC example

- Strongly-connected subsets: $S = \{s_0, s_1\}$, $S' = \{s_1, s_3, s_4\}$, $S'' = \{s_0, s_1, s_3, s_4\}$
- Strongly-connected components: Only $S'' = \{s_0, s_1, s_3, s_4\}$

Checking emptiness

Büchi automaton $A = (Q, s_0, \Sigma, \rightarrow, F)$ with accepting run σ

As Q is finite, there is some suffix σ' of σ s.t. every state on σ' is reachable from any other state on σ'

- I.a.w: those set of states is strongly connected.
- This set is reachable from an initial state and contains an accepting state

Checking non-emptiness of $\mathcal{L}(A)$ is equivalent to finding a SCC in the graph of A that is reachable from an initial state and contains an accepting state

Emptyness checking and counter example

- different algos for SCC. E.g.:
  - Tarjan's version of the *depth-first search* (DFS) algorithm
  - SPIN *nested depth-first search* algorithm
- If the language $\mathcal{L}(A)$ is non-empty, then there is a *counterexample* which can be represented in a finite way
  - It is *ultimately periodic*, i.e., it is of the form $\sigma_1\sigma_2^\omega$, where σ_1 and σ_2 are finite sequences

3.5.2 The algorithm

Model checking algorithm

- Let A be the automaton specifying the system and $\overline{B}$ the automaton corresponding to the negation of the property φ

1. Construct the intersection automaton $C = A \cap \overline{B}$
2. Apply an algorithm to find SCCs reachable from the initial states of C
3. If none of the SCCs found contains an accepting state
   - The model A satisfies the property/specification φ
4. Otherwise,
   a) Take one strongly-connected component SC of C
   b) Construct a path σ_1 from an initial state of C to some accepting state s of SC

c) Construct a cycle from s and back to itself (such cycle exists since SC is a strongly-connected component)
d) Let σ_2 be such cycle, excluding its first state s
e) Announce that $\sigma_1\sigma_2^\omega$ is a counterexample that is accepted by A, but it is not allowed by the property/specification φ

3.5.3 LTL to Büchi

LTL to Büchi

- translation to Generalized Büchi GBA
- cf. Thompson's construction
- *structural* translation
- crucial idea: connect semantics to the syntax.
- compare Hintikka-sets or similar constructions for FOL

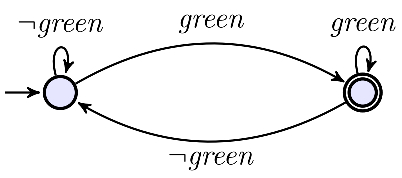
Source and terminology: Baier and Katoen [1]

- transition systems TS:
  - corresponds to Kripke systems
  - state-labelled (transition labels irrelevant)
  - labelled by sets of atomic props: $\Sigma = 2^P$
  - "language" or behavior of the TS: (traces): infinite sequences over Σ

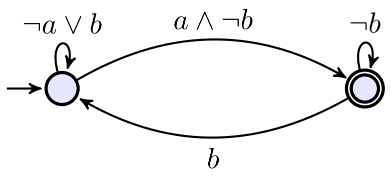
Illustrative examples (5.32)

1. $\Box\Diamond green$
2. $\Box(request \rightarrow \Diamond response)$
3. $\Diamond\Box a$

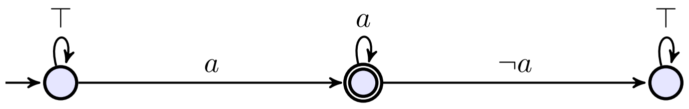
$\Box\Diamond green$



$\Box(\textit{request} \rightarrow \Diamond\textit{response})$



$\Diamond\Box a$



Reminder: Generalized NBA

- equi-expressive than NBA
- used in the construction
- different way of defining acceptance
  - acceptance: set of *acceptance sets* = set of sets of elements of Q.
  - acceptance: each acceptance set F_i must be "hit" infinitely often

Basic idea for $\mathcal{G}_\varphi$

- not the construction yet, but: "insightful" property
- find a mental picture:
  - what are the states of the automaton
  - (and how are they connected by transitions)
- $A_i \in \Sigma$, sets of atomic props
- B_i : "extended" (by sub-formulas of φ), i.e., $B_i \supseteq A_i$.

Namely those that are intended to be in the "language of that state". I.e., the B_i's form the states of $\mathcal{G}_\varphi$.

Given $\sigma = A_0A_1A_2\ldots \in \mathcal{L}(\varphi)$.

Extension to $\hat{\sigma} = B_0B_1B_2\ldots$

$$\psi \in B_i \qquad \text{iff} \qquad \underbrace{A_i, A_{i+1}A_{i+2}\ldots}_{\sigma^i} \models \psi$$

$\hat{\sigma}$ = run (ultimately: state-sequence) in $\mathcal{G}_\varphi$

Rest

Cf. FSAs

- states as “sets” of “words” (language resp. set of ltl formulas)
- cf. Myhill-Nerode
- a bit different, (equivalence on languages of finite words)
- represent states by equivence classes of words

Closure of φ

- related to Fisher-Ladner closure
- See page 276
- “states” A_i from the mental picture
- what’s a “closure” in general?
- Extending A_i to B_i not by *all* true formulas, but only those that could *conceivably play a role* in an automaton checking φ
- $\Rightarrow$ achieving “finiteness” of the construction

How to extend A_i’s

- not by irrelevant stuff (closure of φ).
- two other conditions:
  - avoid contradictions (*consistency*)
  - include logical consequences[1] (*maximality*)
- maximally consistent sets! (here called *elementary*)
- in one state: local perspective only (but don’t forget U)
- Cf: KS has an interpretation for each , here now (in the intended BA), “interpretation” for *all relevant formulas* “in” each state (subformulas of φ and their negation)

Elementary sets/maximally consistent sets

- given φ
- elementary: “maximally consistent set of subsets (of the closure of φ)”
- consistent: “no obvious contradictions”
- maximally consistent: sets for formulas ψ in the closure of φ s.t., there exists *some* path π s.t. $\pi \models \psi$.
  - wrt. propositional logic
  - *locally consistent* wrt. until
- “maximal”

[1]hence the notion of “closure”

Example: $\varphi = a \ U \ (\neg a \wedge b)$

$$\{a, b\} \subseteq \mathit{closure}(\varphi)$$

$$\{a, b, \neg a, \neg b, \neg(\neg a \wedge b), \neg a \wedge b, \varphi, \neg\varphi\}?$$

page 276/277

Example: $\varphi = a \ U \ (\neg a \wedge b)$

$$\sigma = \{a\}\{a, b\}\{b\} \ldots \ = \ A_0 A_1 A_2 \ldots$$

- Extending (for example): A_0 to B_0
- extending σ to $\hat{\sigma}$

Construction of GNBA: general

- given and φ
- given φ, construct an GNBA such that

$$\mathcal{L}(B) = \mathit{words}(\varphi)$$

- 3 core ingredients
  1. states = sets of formulas which (are suppsed to) "hold" in that state
  2. transition relation: connect the states appropriately,
  3. transitions labelled by sets of .
- labeled transition connected states to match the semantics: for $\bigcirc\varphi$: go from a state containing $\bigcirc\varphi$ to a state containing φ. Label the transition with the APs from the start state.

Transition relation

$$\delta : Q \times 2 \rightarrow 2^Q$$

- if $A \neq B\cap$: $\delta(B, A) = \emptyset$
- if $A = B\cap$, then $\delta(B, A)$ is the set B' such that
  - for every $\bigcirc\psi \in \mathit{closure}(\varphi)$:

$$\bigcirc\psi \in B \quad \text{iff} \quad \psi \in B'$$

  - for every $\varphi_1 \ U \ \varphi_2 \in \mathit{closure}(\varphi)$:

$$\begin{array}{lll} \varphi_1 \ U \ \varphi_2 \in B & \text{iff} & \varphi_2 \in B \qquad\qquad\qquad\qquad \text{or} \\ & & (\varphi_1 \in B \quad \text{and} \quad \varphi_1 \ U \ \varphi_2 \in B') \end{array}$$

Accepting states

$$F_{\varphi_1 U \varphi_2} = \{B \in Q \mid \varphi_2 \in B \text{ or } \varphi_1\ U\ \varphi_2 \notin B\}\ .$$

3.5.4 Rest

3.6 Final Remarks

3.6.1 Something on Automata

Kripke Structures and Büchi Automata

Observation

- In Peled's book "Software Reliability Methods" Peled [8] the definition of a Büchi automaton is very similar to our Kripke structure, with the addition of acceptance states
  - There is a labeling of the states associating to each state a set of subsets of propositions (instead of having the propositions as transition labels)
- We have chosen to define Büchi Automata in the way we did since this definition is compatible with the implementation of SPIN
  - It was taken from Holzmann's book "The SPIN Model Checker" Holzmann [5]

Automata Products

Observation

- We have defined synchronous and asynchronous automata products with the aim of using SPIN (based on Holzmann's book)
  - The definition of asynchronous product is intended to capture the idea of (software) asynchronous processes running concurrently
  - The synchronous product is defined between an automaton specifying the concurrent asynchronous processes and an automaton obtained from an LTL formula (or obtained from a Promela `never` claim)
  - The purpose for adding the stutter closure (in the definition of the synchronous product) is to make it possible to verify both properties of finite and infinite sequences with the same algorithm
- I.e., you might find different definitions in the literature!
  - In particular, in Peled's book the automata product is defined differently, since the definition of Büchi automata is different

Further reading

Further reading

- The first two parts of this lecture were mainly based on Chap. 6 of Holzmann's book "The SPIN Model Checker"
  - Automata products: Appendix A
- The 3rd part was taken from Peled's book

For next lecture: Read Chap. 6 of Peled's book, mainly section 6.8 on translating LTL into Automata

- We will see how to apply the algorithm to an example

Bibliography

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

[2] Büchi, J. R. (1960). Weak second-order arithmentic and finite automata. *Zeitschrift für mathematische Logik und Grundlagen der Mathematik*, 6:66–92.

[3] Büchi, J. R. (1962). On a decision method in restricted second-order logic. In *Proceedings of the 1960 Congress on Logic, Methodology and Philosophy of Science*, pages 1–11. Stanford University Press.

[4] Etessami, K., Wilke, T., and Shuller, R. (2001). Fair simulation relations, parity games, and state space reduction for Büchi automata. In *Proceedings of ICALP 2001*, volume 2076 of *Lecture Notes in Computer Science*, pages 153–167. Springer Verlag.

[5] Holzmann, G. J. (2003). *The Spin Model Checker*. Addison-Wesley.

[6] Lamport, L. (1977). Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, 3(2):125–143.

[7] Manna, Z. and Pnueli, A. (1992). *The temporal logic of reactive and concurrent systems—Specification*. Springer Verlag, New York.

[8] Peled, D. (2001). *Software Reliability Methods*. Springer Verlag.

Index