



Course Script

IN 5110: Specification and Verification of Parallel Systems

IN5110, autumn 2021

Martin Steffen

Contents

3	LTL model checking	1
3.1	Linear-time temporal logics	1
3.1.1	Syntax	1
3.1.2	Semantics	2
3.1.3	The past (not pensum)	7
3.1.4	Some LTL examples	8
3.1.5	Dual connectives and complete sets of connectives	11
3.1.6	Classification of properties	13
3.1.7	GCD Example	16
3.2	Automata-based model checking	18
3.2.1	LTL model checking	18
3.2.2	Automata-based LTL model checking	18
3.3	Automata and logic	24
3.3.1	Finite state automata	24
3.3.2	Büchi automata	31
3.3.3	Stuttering	33
3.3.4	From TSs to Büchi	35
3.4	Automata products	38
3.5	Translation LTL to generalized Büchi-automata	44
3.6	GBA to BA	59
3.7	Checking for emptiness	60
3.8	Expressiveness of LTL and complexity	63
3.8.1	Expressiveness of LTL and Büchi automata	63
3.8.2	Complexity	66

3

Chapter 3

LTL model checking

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

$$\dots \longrightarrow \bullet_{i-2}^p \longrightarrow \bullet_{i-1}^p \longrightarrow \bullet_i^p \longrightarrow \bullet_{i+1}^p \longrightarrow \bullet_{i+2}^p \longrightarrow \dots$$

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.

3.1.1 Syntax

As before, we start with the **syntax** of the logic at hand, and 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.

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.

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

Table 3.1: LTL syntax

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 3.1.1 (Comparison to modal logics). 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. $\square$

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 examples.

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 $p U q \vee \Box p$.

3.1.2 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

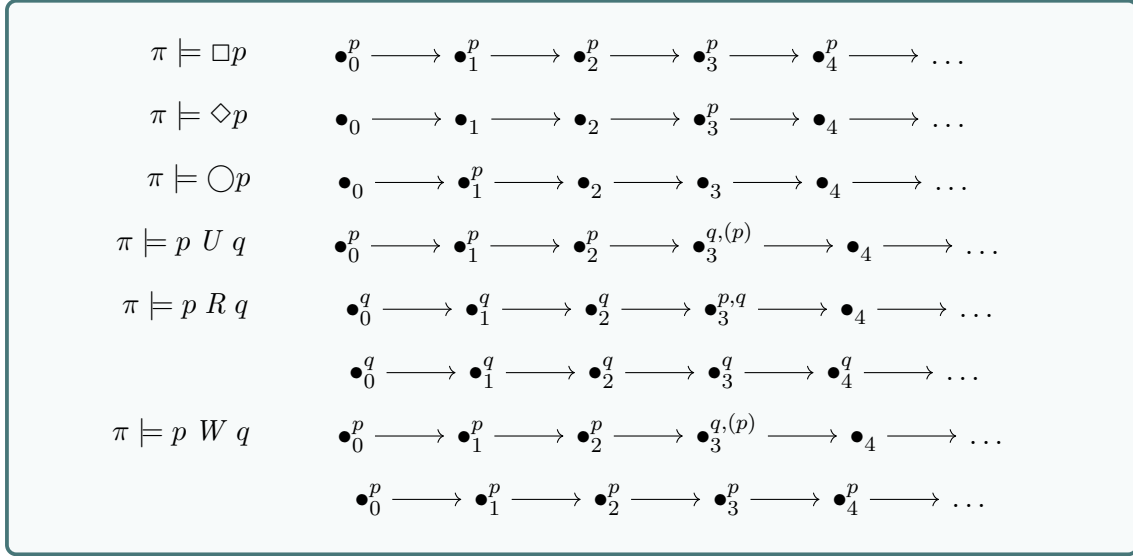


Figure 3.1: Illustration of LTL formulas

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 $AP \rightarrow \mathbb{B}$ 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** (a path here)¹ then provides a state per world, i.e., a mapping

$$\mathbb{N} \rightarrow (AP \rightarrow \mathbb{B}) . \quad (3.1)$$

This is equivalently represented as an infinite sequence of the form

$$s_0 s_1 s_2 \dots \quad (3.2)$$

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**.

¹Later we will encounter transitions systems, basically graphs, i.e., non-linear constructions. They can also seen as Kripke structures, because we also discuss what it means that such a structure satisfies an LTL-formula; i.e., we see also the transition system as a model of for LTL formulas. The connection is very simple, a transition system with some initial states satisfies a formula when when all the paths through the transition system starting at some initial state satisfies the formula.

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 infinite 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” implicit 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 transitions or edges in the underlying transition system or graph.

Finally, of course, the conventional notion of path in a graph does not speak 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 $\mathbb{N}_n \rightarrow N$ 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 3.1.2 is an infinite sequence of states (or valuations).

Paths and computations

Definition 3.1.2 (Path). A *path* is an infinite sequence

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

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

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

It's intended that (later) paths represent behavior of programs resp. “going” through a transition system. A transition 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 graphs². Note, however, that we have defined paths **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 states (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 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 $\models_{ul}$ available, that works on *states*. Note that in case of first-order logic, a signature and its *interpretation* is assumed to be fixed.

Definition 3.1.3 (Satisfaction). A path π *satisfies* an LTL formula φ , written $\pi \models \varphi$, under the following conditions:

$\pi \models \psi$	iff	$\pi_0 \models_{ul} \psi$ with ψ from the underlying core language
$\pi \models \neg\varphi$	iff	$\pi \not\models \varphi$
$\pi \models \varphi_1 \wedge \varphi_2$	iff	$\pi \models \varphi_1$ and $\pi \models \varphi_2$
$\pi \models \bigcirc\varphi$	iff	$\pi^1 \models \varphi$
$\pi \models \varphi_1 \, U \, \varphi_2$	iff	$\pi^k \models \varphi_2$ for some $k \geq 0$, and $\pi^i \models \varphi_1$ for every i such that $0 \leq i < k$

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-expanded 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.

²Remember that the mathematical notions of graph corresponds to Kripke frames

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 “releases 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 3.1.4 (Semantics of derived operators).

$$\begin{aligned} \pi \models \Box \varphi & \quad \text{iff } \pi^k \models \varphi \text{ for all } k \geq 0 \\ \pi \models \Diamond \varphi & \quad \text{iff } \pi^k \models \varphi \text{ for some } k \geq 0 \\ \pi \models \varphi_1 R \varphi_2 & \quad \text{iff for every } j \geq 0, \\ & \quad \text{if } \pi^i \not\models \varphi_1 \text{ for every } i < j \text{ then } \pi^j \models \varphi_2 \\ \pi \models \varphi_1 W \varphi_2 & \quad \text{iff } \pi \models \varphi_1 U \varphi_2 \text{ or } \pi \models \Box \varphi_1 \end{aligned}$$

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 3.1.5 (Validity and equivalence).

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

Example 3.1.6. $\Box$ distributes over $\wedge$, while $\Diamond$ distributes over $\vee$.

$$\begin{aligned} \Box(\varphi \wedge \varphi_2) & \sim (\Box \varphi \wedge \Box \varphi_2) \\ \Diamond(\varphi_1 \vee \varphi_2) & \sim (\Diamond \varphi_1 \vee \Diamond \varphi_2) \end{aligned}$$

□

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 or inconsistent formula (one that is equivalent to $\perp$), as it holds for all systems, no matter what (resp. $\perp$ holds for no system).

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

3.1.3 The past (not pensum)

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 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 are 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 $\mathbb{N} \rightarrow (AP \rightarrow \mathbb{B})$), 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” [29]. Let’s write $\Box^{-1}$, $\Diamond^{-1}$ etc. for **past operators**. The definition of the satisfaction relation is straightforward

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

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

$$\Box(\varphi \rightarrow \Diamond^{-1}\psi) \quad (3.3)$$

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:

$$\psi R (\varphi \rightarrow \psi) . \quad (3.4)$$

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

Figure 3.2 illustrates the two equivalent formulations of the property.

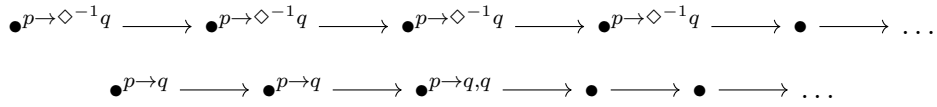


Figure 3.2: Illustration of the formulas from Example 3.1.7

3.1.4 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, though, one starts with informal requirements and it can be difficult to correctly express informally stated requirements.

Let's try to capture the vaguely formulated property

“when p then q .”

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

- $\varphi \rightarrow \psi$? $\varphi \rightarrow \psi$ holds in the initial state.
- $\Box(\varphi \rightarrow \psi)$? $\varphi \rightarrow \psi$ holds in every state.
- $\varphi \rightarrow \Diamond\psi$? φ holds in the initial state, ψ will hold in some state.
- $\Box(\varphi \rightarrow \Diamond\psi)$? “response”

The last formulation is also called a **response** property (and sometimes one uses a special notation for that, namely $\varphi \rightsquigarrow \psi$, see Example 3.1.9). It is not obvious, which one of them (if any) is necessarily what is intended.

Let's do a few more examples.

Example 3.1.8. $\varphi \rightarrow \Diamond\psi$: If φ holds initially, then ψ holds eventually.



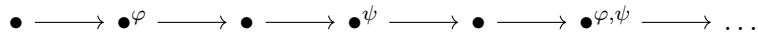
This formula will also hold in every path where φ does not hold initially.



□

Example 3.1.9 (Response). $\Box(\varphi \rightarrow \Diamond\psi)$

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



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



□

Example 3.1.10 (∞). $\Box\Diamond\psi$ There are infinitely many ψ -positions.



□

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

Before we continue with even 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 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 3.1.10 states that there are infinitely many 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.

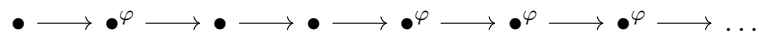
Obviously, properties can contain both aspects, like in a conjunction of $\Box\varphi$ and one of the three previous examples. However, it can be shown that every LTL formula can be equivalently expressed by a conjunction of a pure safety property and a pure liveness properties. We will talk more precisely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

The discussion here reasoned that liveness properties like the one from Example 3.1.10 are intuitively more complex, by pointing out that they intuitively 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 . If so, the property does not hold, if one finds no violation, the property holds. So that's a straightforward graph search. Of course 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 3.1.10 by a finite system. We will see that later, when we address the algorithmic problem to LTL model checking.

But 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).

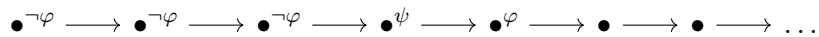
Example 3.1.11 (Permanence). Eventually φ will hold *permanently*: $\Diamond\Box\varphi$.



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

Example 3.1.12. $(\neg\varphi) W \psi$

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



φ may never hold

$$\bullet \neg\varphi \longrightarrow \bullet \neg\varphi \longrightarrow \bullet \neg\varphi \longrightarrow \bullet \neg\varphi \longrightarrow \bullet \neg\varphi \longrightarrow \bullet \neg\varphi \longrightarrow \bullet \neg\varphi \longrightarrow \dots$$

□

Example 3.1.13. $\Box(\varphi \rightarrow \psi \ W \ \chi)$

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

$$\bullet \longrightarrow \bullet \varphi, \psi \longrightarrow \bullet \psi \longrightarrow \bullet \psi \longrightarrow \bullet \psi \longrightarrow \bullet \chi \longrightarrow \bullet \longrightarrow \bullet \varphi, \psi \longrightarrow \dots$$

The sequence of ψ -positions need not terminate.

$$\bullet \longrightarrow \bullet \varphi, \psi \longrightarrow \bullet \psi \longrightarrow \bullet \psi \longrightarrow \bullet \psi \longrightarrow \bullet \psi \longrightarrow \bullet \psi \longrightarrow \bullet \psi \longrightarrow \dots$$

□

3.1.5 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 3.1.14 (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 $\nrightarrow$ for the negated reverse implication, then that is the dual of $\rightarrow$:

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

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., $\not\leftrightarrow$ can be defined), but also subsets of it, so we don't actually need all the connectives.

Example 3.1.15. 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:

$$\begin{aligned}
\varphi \rightarrow \psi &\sim \neg\varphi \vee \psi \\
\varphi \leftrightarrow \psi &\sim (\varphi \rightarrow \psi) \wedge (\psi \rightarrow \varphi) \\
\top &\sim p \vee \neg p \\
\perp &\sim p \wedge \neg p
\end{aligned}$$

□

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 losing 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, and $\bigcirc$ is its own dual:

$$\neg\Box\varphi \sim \Diamond\neg\varphi \quad \text{and} \quad \neg\Diamond\varphi \sim \Box\neg\varphi \quad \text{and} \quad \neg\bigcirc\varphi \sim \bigcirc\neg\varphi \quad (3.5)$$

As a side remark: the symbols $\Box$ and $\Diamond$ are also part of modal logics in general, with a different interpretation (there we called them generally “necessity” and “possibility”). 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.

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

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:

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

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

invariant	$\Box\varphi$
example of a liveness property	$\Diamond\varphi$
obligation	$\Box\varphi \vee \Diamond\psi$
recurrence	$\Box\Diamond\varphi$
persistence	$\Diamond\Box\varphi$
reactivity	$\Box\Diamond\varphi \vee \Diamond\Box\psi$
fairness	different forms of fairness

Table 3.2: Classification of LTL properties

3.1.6 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 role 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: as said, 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 pure safety properties nor pure lifeness properties. However, there is a result that states, any LTL property could equivalently be written as conjunction of a pure liveness property and a pure safety property.

Safety properties: never anything bad happens

Table 3.2 contains a few kind of formulas, where we can assume that 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.

An **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 [2]. It’s however true that **invariants** are perhaps the most typical, easiest, and most 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 Leslie Lamport. We also focus on invariance (see Definition 3.1.16), without giving the slightly more complex general 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 3.1.16 (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 3.1.16 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 3.1.17 (Mutual exclusion). *Mutual exclusion* is a safety property. Let c_i denote that process P_i is executing in the critical section. Then

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

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

$$\Diamond(c_1 \wedge c_2)$$

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 3.1.18 (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.

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

Here the definition of obligations.

Definition 3.1.19 (Obligation). A *simple obligation* formula is of the form

$$\Box\varphi \vee \Diamond\psi$$

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

Such a property can equivalently be written as implication

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

where $\varphi' = \neg\varphi$. The equivalent implicational form $\Diamond\varphi \rightarrow \Diamond\psi$ states that if some state satisfies χ , then some state must satisfy ψ , which perhaps makes the terminology “obligation” a bit clearer. Obligations actually subsume safety and liveness properties (in 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.

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

and the facts that $\models \neg\Box\perp$ and $\models \neg\Diamond\perp$.

□

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

Definition 3.1.20 (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.

3.1.7 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;
      P1 :: [ l0 : [ l1 : while x ≠ y do l2 : [
                [ l3 : await x > y; l4 : x := x - y; ]
                or
                [ l5 : await y > x; l6 : y := y - x; ] ]
                l7 : g := x; l8 : ] ] ]

```

ms
Error:
Other
code

ms Error: In the slides, we used this code:

```

11 while x != y do
12   if x > y do
13     x := x-y
14   else if x < y do
15     y := y-x
16   fi
17 od
18 g := x
19

```

But the text needs to be adapted as well.

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 exactly 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 from the infinite execution). The GCD program is terminating, so it does not technically continue 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 $at(l_8)$.

$$\begin{aligned} \pi : \quad & \langle l_1, 21, 49, 0 \rangle \rightarrow \langle l_2^b, 21, 49, 0 \rangle \rightarrow \langle l_6, 21, 49, 0 \rangle \rightarrow \\ & \langle l_1, 21, 28, 0 \rangle \rightarrow \langle l_2^b, 21, 28, 0 \rangle \rightarrow \langle l_6, 21, 28, 0 \rangle \rightarrow \\ & \langle l_1, 21, 7, 0 \rangle \rightarrow \langle l_2^a, 21, 7, 0 \rangle \rightarrow \langle l_4, 21, 7, 0 \rangle \rightarrow \\ & \langle l_1, 14, 7, 0 \rangle \rightarrow \langle l_2^a, 14, 7, 0 \rangle \rightarrow \langle l_4, 14, 7, 0 \rangle \rightarrow \\ & \langle l_1, 7, 7, 0 \rangle \rightarrow \langle l_7, 7, 7, 0 \rangle \rightarrow \langle l_8, 7, 7, 7 \rangle \rightarrow \dots \end{aligned}$$

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

$\Box terminated$	(safety)
$at(l_1) \rightarrow terminated$	
$at(l_8) \rightarrow terminated$	
$at(l_7) \rightarrow \Diamond terminated$	(conditional liveness)
$\Diamond at(l_7) \rightarrow \Diamond terminated$	(obligation)
$\Box(gcd(x, y) \doteq gcd(a, b))$	(safety)
$\Diamond terminated$	(liveness)
$\Diamond \Box(y \doteq gcd(a, b))$	(persistence)
$\Box \Diamond terminated$	(recurrence)

3.2 Automata-based model checking

3.2.1 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

$$M \models^? \varphi, \quad (3.9)$$

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 (or starting states), and the transitions represent steps of a program or system (resp. perhaps an abstraction of a real system). So, since the model will be a transition system, we will use T instead of M in the following.³

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

3.2.2 Automata-based LTL model checking

As variously mentioned, 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** LTL 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).

ms Error: Port it to transition systems. Now, in the setting of LTL, with transition systems as models, and given the model-checking problem from equation (3.9), how could one go for it?

³The definition of $\models$ for LTL given earlier was defined for paths, i.e., we defined $\pi \models \varphi$, and it would therefor be correct to call a path π a model of φ . But we are interested in $T \models \varphi$, which means all paths from T (starting at some initial state) satisfy φ , and we consider T as the model: we are interested in checking the behavior of a program, i.e., *all* its behavior, not one particular run.

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 (3.9) holds, if all paths in T , that start from some initial state s_0 , satisfy p (see Definition 3.1.3). 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., $T \models \Box p$ 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 (apart from $\bigcirc p$), 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 T , 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 search 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 $\perp$ (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:

$$\models \varphi \quad \text{iff} \quad \neg\varphi \models \perp . \quad (3.10)$$

Remember, that the *core* of the satisfaction relation for LTL is defined as a relation between one paths and one formulas (see Definition 3.1.3). The satisfaction relation between transition systems and formulas from equation (3.9) is *derived* from that, posing a requirement on *all* paths of T starting its initial state.

Formulas describe sets of paths, namely all paths that make the formula true, but likewise transition systems describes sets of paths, namely all the paths that start in its initial state. We can therefore view the model checking problem $T \models^? \varphi$ from equation (3.9) as entailment problem.

Let's treat the transition system T as specification of all its paths. We can write $\llbracket T \rrbracket$ 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 T ”, i.e. writing $\pi \models T$ for $\pi \in \llbracket T \rrbracket$. With this picture in mind, then the model-checking question $T \models^? \varphi$ is the logical entailment: every path “satisfying” T 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 (3.10), the entailment can be refuted based on the following equivalence:

$$\Gamma \models \varphi \quad \text{iff} \quad \Gamma, \neg\varphi \models \perp . \quad (3.11)$$

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

$$T \models \varphi \quad \text{iff} \quad T \wedge \neg\varphi \models \perp . \quad (3.12)$$

Alternatively we can formulate entailment as subset requirement:

$$\llbracket T \rrbracket_{\mathcal{L}} \subseteq \llbracket \varphi \rrbracket_{\mathcal{L}} \quad \text{iff} \quad \llbracket T \rrbracket_{\mathcal{L}} \cap \overline{\llbracket \varphi \rrbracket_{\mathcal{L}}} = \emptyset . \quad (3.13)$$

where we use $\overline{A}$ to represent the complement of a set A . So $\overline{\llbracket \varphi \rrbracket_{\mathcal{L}}}$ is represents the negated formula $\llbracket \neg\varphi \rrbracket_{\mathcal{L}}$.

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 it into an algorithm. So how can one solve the problem

$$\llbracket T \rrbracket \cap \overline{\llbracket \varphi \rrbracket} = \emptyset, \quad (3.14)$$

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

In the refutation-discussion we said, the T can also be seen as a “logical” specification, namely specifying a set of paths. But of course T and φ are specifications of paths of quite different formalisms. T is a transition system and φ a logical formula. So a notation like $\varphi \wedge T$ we allowed ourselves (in equation (3.12)) is a bit dubious (but of course the $\wedge$ is undoubtedly explained by the intersection formulation from equation (3.13)).

To check entailment resp. the refutation formulation from equation (3.13) 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 is *included* in the other. For the refutation formulation, we need to do to

1. calculate two infinite sets of infinite paths, $\llbracket T \rrbracket$ and $\llbracket \varphi \rrbracket$
2. calculate the complement of the latter,
3. calculate the intersection between them,
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** or descriptions 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 T and the formula φ .

Unfortunately, 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 they both describe infinite sets of paths.

To put them on a common level, we 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 finite representations of the infinite sets of paths are

1. translate $\neg\varphi$ to $\mathcal{A}_{\neg\varphi}$, a finite representation 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 T by translating it to a transition system. In principle that's correct, but there is fine-print. The fine print is that the translated $\mathcal{A}_{\neg\varphi}$ is not exactly of the same form as the transition system T . We will see that when looking in detail at the construction (but gloss over the details now, when discussing the big picture). Indeed $\mathcal{A}_{\neg\varphi}$ 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 the relevant information resp. are labelled. For the automata, it's the *edges* which are labelled (as is standard for automata).⁴ 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-labelled and node-labelled, are interchangeable, 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 parallel 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 representation.

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 than regular expressions (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 **infinite-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

⁴Our definition of transition systems also attaches information to states. However, for the content of the chapter and the discussion here, this information is not relevant. Since there exists nothing that does not exist: there are also FSA where both states and edges are labelled. For instance so-called Moore machines which are a form of input/output automata, but we don't work with them.

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-flavors which are used 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 acceptance 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 acceptance condition has changed, the implementation is unchanged except when to stop generating or accepting a word.

Indeed, since acceptance 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 automaton 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 representations.

So what actually needs to be solved algorithmically is

- translation:** find a translation for LTL formulas to an equivalent Büchi automaton
- intersection:** construct an automata that represents the intersection of two automata
- emptiness:** check if the language of a given automata is empty

As said, a construction covering complementation can be avoided, we simply translate $\neg\varphi$. The overall construction is summarized in Figure 3.3.

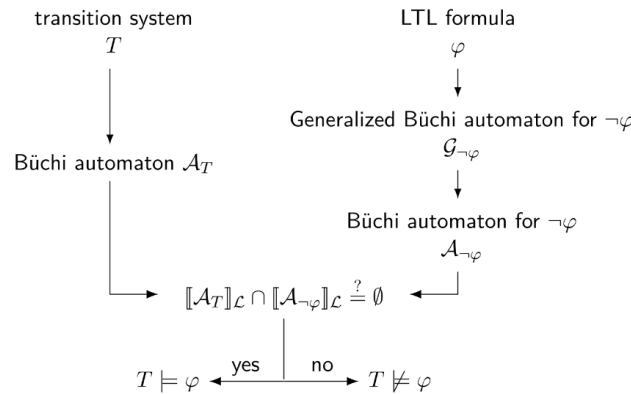


Figure 3.3: Automata-based LTL model checking

We see that the translation from LTL to Büchi-automata is done in 2 steps: we don't go directly to a Büchi-automaton, but translate first to a variation thereof, known as generalized BA (simply because it's easier to do it in two steps).

ms
Note:
Tikz

ms Note: The figure does not $\LaTeX$ well in the script (only in the slides). Try to fix that. currently it's just a png-include.

3.3 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 standard finite-state automata, and afterwards Büchi-automata, a well-known representative of finite-state automata for infinite words.

3.3.1 Finite state automata

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

Definition 3.3.1 (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 automaton 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 philosophical context, the *edges* between the worlds may not be viewed as *transitions* or operational steps in a evolving system. In Baier and Katoen [2], they call Kripke structures *transition systems* (actually without even mentioning Kripke structures).

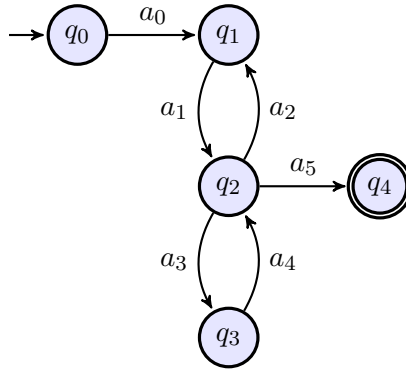
We are not obsessed with terminology. But as preview for later: In the central construction about model checking LTL, the system on the one hand will be represented as a (finite) transition system where the *states* are labelled and the LTL formula on the other hand will be represented by an *automaton* whose *transitions* are labelled. The automaton will be called *Büchi-automaton*. The definition corresponds to the one just given in Definition 3.3.1. What makes it “Büchi” is not the form or data structure of the automaton itself, it the *acceptance* condition, i.e., the interpretation of the set of accepting states.

Let’s illustrate the definition on a small example:

Example 3.3.2. The automaton is given by the 6-letter alphabet (or label set) $\Sigma = \{a_0, a_1, \dots, a_5\}$, by the 5 states $q_0, q_1, \dots, q_4$, 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

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

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* (see Figure 3.4).

Determinism vs. non-determinism

Determinism in a system generally 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

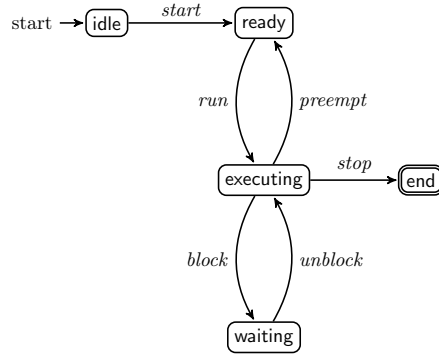


Figure 3.4: FSA scheduler

from the alphabet, there are at most one successor states. The automata from Definition 3.3.1 can be non-deterministic, the definition is based on a transition *relation*.

Definition 3.3.3 (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, which 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 automata 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 $A \rightarrow B_\perp$, 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 languages of automata

Definition 3.3.4 (Run). A *run* of a finite state automaton $\mathcal{A} = (Q, q_0, \Sigma, F, \rightarrow)$ is a (possibly infinite) sequence

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

The notation $q \xrightarrow{a} q'$ is meant as $(q, a, q') \in \rightarrow$. 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 [22] (the “Spin-book”). The book Baier and Katoen [2], 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.

Side remark 3.3.5 (Other views on the concept of execution). 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 “truly concurrent”). Also in connection with weak memory models, such relaxations are common. Considering partial orders (when it fits) is also a optimization technique: by avoiding 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). $\square$

Example 3.3.6 (Run). Consider the automaton from Figure 3.4. State sequences arising from possible runs look like

idle ready (executing 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 correspond to more than one word, but in the example, the transition-label sequence determines the state sequence: the automaton is deterministic. $\square$

Definition 3.3.7 (Acceptance). An *accepting* run of a finite state automaton $\mathcal{A} = (Q, q_0, \Sigma, F, \rightarrow)$ is a *finite* run $\sigma = q_0 \xrightarrow{a_0} q_1 \xrightarrow{a_1} \dots \xrightarrow{a_{n-1}} q_n$, with $q_n \in F$.

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

idle ready executing waiting executing end .

The corresponding *word* of labels is

start run block unblock stop .

An 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 existence 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 accepts 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 After this digression, back to the main course of the section. Let's just define what it means that an automaton accepts a word. That also defines the *language* of an automaton as the set of all accepted words.

Definition 3.3.8 (Language). The *language* $\llbracket \mathcal{A} \rrbracket_{\mathcal{L}}$ (sometimes also written $\mathcal{L}(\mathcal{A})$) 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 expressible 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

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

We use $\mid$ 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 languages 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 at 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 .

That property can be represented by the automaton from Figure 3.5.

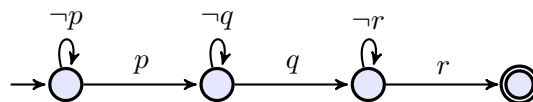


Figure 3.5: Automaton for the negated specification

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 existence of a finite run.

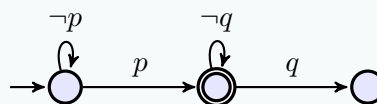
That fits to the standard informal explanation of the concept, stipulating: “that never something bad happens” (because if some bad thing happens, it means that one can detect it in a finite amount of time). The slogan is attributed to Lamport [26]. That “bad” in the sentence refers to the *negation* of the original property one wishes to be established (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 that has the following (meta-)property: If the safety property holds *for all finite behavior*, then it holds *for all behavior* (all behavior includes *infinite* 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 automata to reason about infinite run?

Let’s also have a look at a liveness 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 visiting states *infinitely* often.

In the above example, acceptance could be to “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.

3.3.2 Büchi automata

As mention, Büchi-automata are defined as finite state automata from Definition 3.3.1. What makes them “Büchi” is the different acceptance condition, that can handle infinite words or runs. Infinite runs are often called ω -runs (“omega runs”), and the corresponding acceptance condition consequently ω -acceptance. There are different versions of the idea (Büchi, Muller, Rabin, Streett, parity etc.), but here we present **Büchi acceptance** [9] [8].

Definition 3.3.9 (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**.

Example 3.3.10 (Infinite runs). Consider the automaton from Figure 3.6. It almost the same than the one from Figure 3.4 earlier except that the accepting state has been changed.

One accepted infinite state sequence is the following:

idle (ready executing) $^\omega$

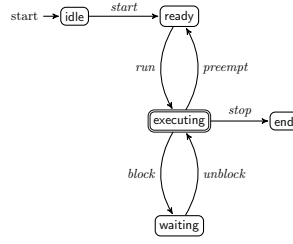


Figure 3.6: Scheduler (2)

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 than 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 mentioned 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 $\text{inf}(\sigma)$ 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 3.3.11 (Generalized Büchi automaton). A *generalized Büchi automaton* is an automaton $\mathcal{A} = (Q, q_0, \Sigma, \mathcal{F}, \rightarrow)$, where $\mathcal{F} \subseteq 2^Q$.
Let $\mathcal{F} = \{f_1, \dots, f_n\}$ and $f_i \subseteq Q$. A run σ of $\mathcal{A}$ is *accepting* if

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

Note that not only the acceptance has changed, but also the *format* of the automaton. Büchi automata, as standard finite state automata, the acceptance is based on a set F of states. Here, $\mathcal{F}$ is not a set of states, $\mathcal{F}$ is a set of sets 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 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 automata is also known as ω -regular languages.

3.3.3 Stuttering

Next we address an issue not so much from automata theory, but motivated by the use we make of the concepts modelling systems and model checking them. In the big-picture discussion earlier we mentioned that the approach works with two rather similar representations with nodes and edges and labels, namely automata and transition systems. They are pretty close, still different in some details, 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, programs may **terminate**.

So that's another mismatch, LTL works on **infinite runs**, and Büchi acceptance, generalized 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) terminates, we simply let it artificially

continue infinitely by doing nothing. That is known as **stuttering**. This allows to treat finite and infinite acceptance uniformly by Büchi-acceptance condition. It avoids coming up with a 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 infinite one, where, at some point, infinitely often ε is done. Stuttering (here) is allowed only at the end, i.e. a run must end in an end-state

Definition 3.3.12 (Stutter extension). The *stutter extension* of a finite run σ with last state q_n , is the ω -run

$$\sigma (q_n, \varepsilon, q_n)^\omega . \quad (3.15)$$

Let's revisit the scheduler example again

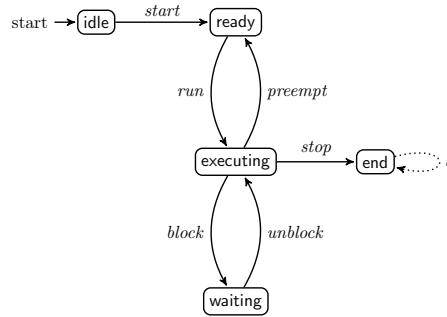


Figure 3.7: Scheduler with stuttering

Example 3.3.13 (Stuttering). The “process scheduler” example from Figure 3.4 uses the accepting state *end* now as natural end state with an ε -loop which is also accepting (see Figure 3.7). Examples of accepting state sequences resp. accepting words corresponding to an accepting ω -run include the following:

idle ready executing waiting executing end $^\omega$

and

start run block unblock stop $^\omega$

□

So far, we have introduced the stutter extension of a (finite) 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 be 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.

3.3.4 From TSs to Büchi

In this section we bridge the 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

From transition systems 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. As stated, we don’t use the traditional terminology any longer in this chapter, we speak about transition system. So let’s fix their definition, before we continue.

Definition 3.3.14 (Labeled Transition System). A *labelled transition system* T is a tuple $(S, \Sigma, \rightarrow, I, AP, L)$ where

- S is a set of states.
- Σ is a set of actions.
- $\rightarrow \subseteq S \times \Sigma \times S$ is the transition relation
- $I \subseteq S$ is a set of initial states.
- AP is a set of atomic propositions.
- $L : S \rightarrow 2^{AP}$ is a labeling function.

T is called *finite* if S, Σ , and AP are finite.

Remark 3.3.15 (Labelling). It should be noted that the definition of labelled transition systems is **specific** for the purpose here, in the context of automata-based model checking. What is specific is the form of the labelling: the function L labels states with sets of atomic proposition. The notion of labelled transition systems occurs also outside the role they play here, with other information attached to states and/or transitions. Indeed, when talking about *labelled* transition systems, often it’s the transitions which are meant to be labelled. It’s just another situation when terminology is not universally agreed upon, or the same word is used for different definitions, but actually conceptually, one means similar things.

Our definition of LTSes labels states as well as transitions (though the labelling of the transitions is not captured by a labelling function). For the constructions in connection with LTL, actually only the state labelling is relevant. $\square$

The transition systems are here 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 [2] 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 satisfaction 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 check that the “language” of the system T is a subset of the language described by a LTL-specification φ , like $T \models \varphi$ corresponds to $\llbracket T \rrbracket_{\mathcal{L}} \subseteq \llbracket \varphi \rrbracket_{\mathcal{L}}$. 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, $\llbracket T \rrbracket_{\mathcal{L}}$ is a language of infinite words corresponding to sequences of “states” and the state-attached information. On the other hand, $\llbracket \varphi \rrbracket_{\mathcal{L}}$ is a language containing words referring to edge-labels of an 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 doing 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 straightforward). 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 description 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.

Definition 3.3.16 (Translating a transition system into an Büchi-automaton). Given $T = (Q, \Sigma, \rightarrow, I, AP, L)$. The corresponding Büchi automaton is defined as $\mathcal{A} = (Q', q_0, \Sigma', F, \rightarrow)$ where

transition labels: $\Sigma' = 2^{AP}$

states:

- $Q' = Q + \{i\}$
- $q_0 = i$
- $F = Q + \{i\}$

transitions:

- $q \xrightarrow{a} q'$ iff $q \Rightarrow_T q'$ and $a = L(q')$ and $q, q' \in Q$
- $i \xrightarrow{a} q$ iff $q \in I$ and $a = L(q)$

Note: the translation turns *all* states of the transition system into **accepting states** of the Büchi automaton. Basically, the accepting conditions are not really “interesting”, and making all states accepting means: I am interested in *all* behavior as long as it’s infinite. The TS (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, continue infinitely anyway by *stuttering*).

Note also, that the valuations $L : S \rightarrow (AP \rightarrow \mathbb{B})$ attach to each state a mapping, that assigns to each atomic proposition from AP 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 $W \rightarrow 2^{AP}$. Perhaps confusingly the assignment of (here Boolean values) to atomic propositions, i.e., functions of type $AP \rightarrow \mathbb{B}$, 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 3.3.17 (From Kripke structure to Büchi automata). The Kripke structure from Figure 3.8a is translated to the Büchi-automaton from Figure 3.8b. BTW: the (only) infinite run of the Kripke structure satisfies (for instance) $\Box q$ and $\Box \Diamond p$.

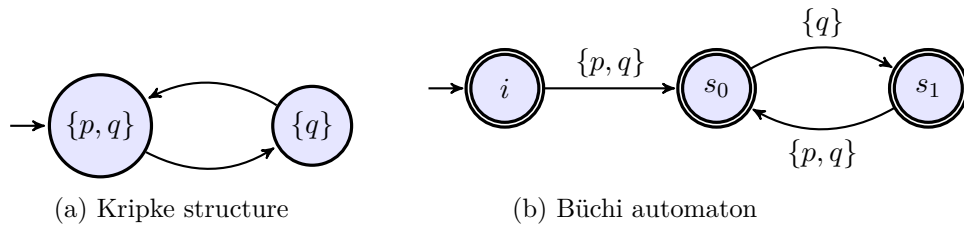


Figure 3.8: Translation

□

3.4 Automata products

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

$$\llbracket \mathcal{A}_T \rrbracket_{\mathcal{L}} \cap \llbracket \mathcal{A}_{\neg\varphi} \rrbracket \quad (3.16)$$

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

The construction 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 (3.16). In particular, the T 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 acceptance 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 T 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 symmetric, i.e., commutative).

The *asynchronous* product here comes from the underlying computation model. The presentation here is based on [22] and thus influenced by the choices as made in the Spin model checker. On the other hand, what Spin does is conventional and common. 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 synchronization by channels, processes do their steps independently, i.e., it's a fairly conventional **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 independent or **asynchronous** manner (though synchronization when needed can be achieved 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 ‘synchronously’ in parallel (with the clock as global synchronizing mechanism). And in such a setting, parallel composition rephrases 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 communicate 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).

Asynchronous product (of automata)

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

Definition 3.4.1 (Asynchronous product). The *asynchronous product* of two automata A_1 and A_2 , (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
- $F = \{(q_1, q_2) \mid q_1 \in F_1 \text{ or } q_2 \in F_2\}$.

$$\frac{q_1 \rightarrow_1 q'_1}{(q_1, q_2) \rightarrow (q'_1, q_2)} \text{PAR}_1 \qquad \frac{q_2 \rightarrow_2 q'_2}{(q_1, q_2) \rightarrow (q_1, q'_2)} \text{PAR}_2$$

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 of 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 automaton, 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.

Synchronous product of Büchi-automata

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, but not overly so.

The general intention of such a “product” construction is that the 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 common 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, or wait for the discussion of the issue later in Example 3.4.4).

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 [22] 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 synchronous product is slightly more elegant compared to standard BAs.

Definition 3.4.2 (Synchronous product (special case)). The *synchronous* product of two finite Büchi automata $\mathcal{B}_1$ and $\mathcal{B}_2$ (written $\mathcal{B}_1 \otimes \mathcal{B}_2$), for the *special case* where $F_1 = Q_1$, is defined as Büchi automaton $\mathcal{B} = (Q, q_0, \Sigma, F, \rightarrow)$ where:

- $Q = Q_1 \times Q_2$
- $q_0 = (q_{01}, q_{02})$
-

$$\frac{q_1 \xrightarrow{a} q'_1 \text{ and } q_2 \xrightarrow{a} q'_2}{(q_1, q_2) \xrightarrow{a} (q'_1, q'_2)}$$

- $(q_1, q_2) \in F$ if $q_2 \in F_2$

Example 3.4.3 (Synchronous product of Büchi automata (special case)). Let's illustrate the definition by a small example (see Figure 3.9). □

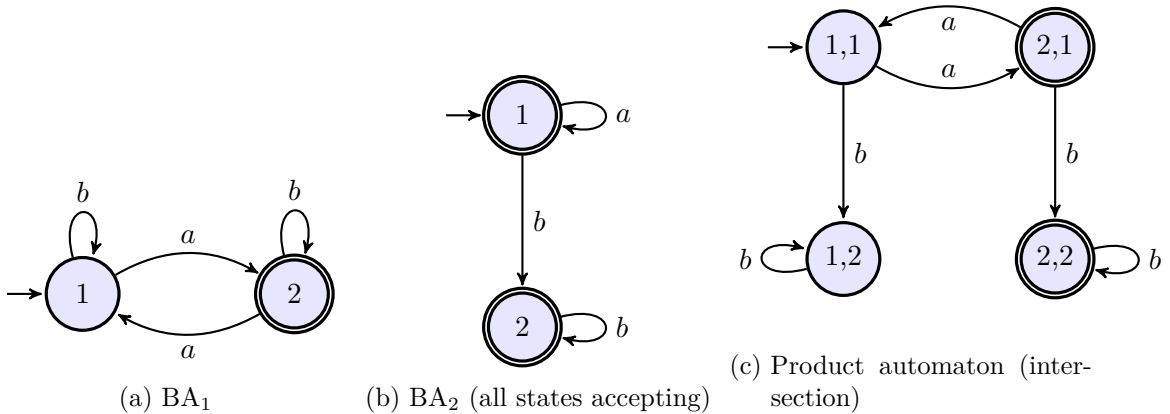


Figure 3.9: Synchronous product of 2 BAs (special case)

We have what we need, namely the definition of synchronous product in the special situation of our model checking problem. But let's round off the section by seeing why defining

the synchronous product of BAs is not straightforward, and how a doing it properly leads to the definition of generalized Büchi automata.

Example 3.4.4 (Synchronous product of BAs and motivation for generalized BAs). The example here illustrates synchronous products of automata, in particular that it's more complex for BAs compared to standard automata. Remember that the difference between automata and BAs is the notion of **acceptance**, the basic machine is the same. Figure 3.10a and 3.10b show two automata, which can be interpreted as standard FSA or alternatively as BAs. The two machines have the same structure (“same” states and transitions), and differ only in their accepting states. In addition, the two machines are deterministic. This is mainly for the purpose of making the example simpler.

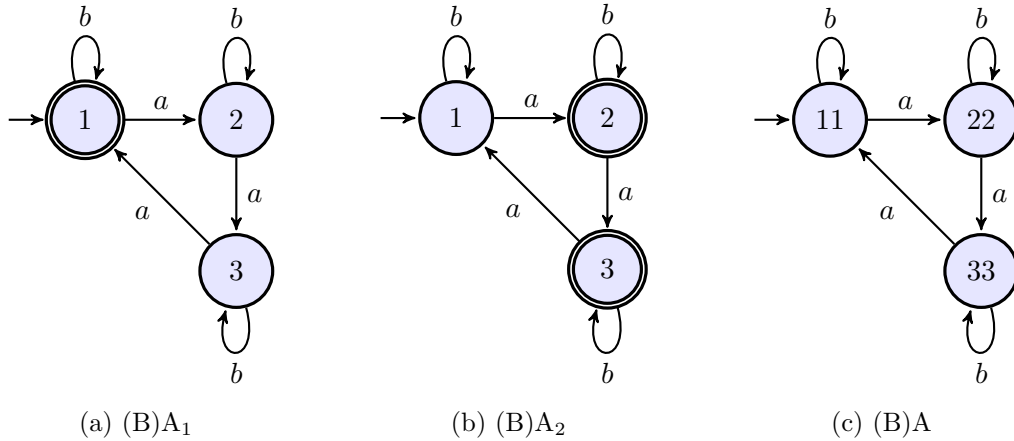


Figure 3.10: Synchronous product

The standard synchronous product is shown in Figure 3.10c, and since the example is so simple —two analogous and deterministic machines are put together— the result looks analogous again: three states, connected with the same transitions and with the “same” initial state. With the accepting states the only difference between the two original machines, and with acceptance as the only difference between FSAs and BAs, the accepting states is where we should focus attention on.

The standard definition of synchronous product of FSAs defines the accepting states of the result as follows: the product automaton is in an accepting state, if both component automata are one of their respective accepting states. It's a sort of conjunction of the acceptance condition. This corresponds to the resulting machine in Figure 3.10c. The resulting automaton has **no** accepting state. As a consequence, the accepting language is **empty**. For finite state automata, that is correct. The first automaton accepts languages with $3n$ occurrences of a s, the second one with $3n + 1$ or $3n + 2$ occurrences of a s. There is thus no word in both languages, the intersection is empty.

If one uses **Büchi-acceptance** instead, things **change**. The languages of both automata are more or less infinite words with infinitely many a 's or infinitely many b 's, so their intersection should have likewise the same language.⁵ However, the product automaton

⁵This characterization of the languages is not 100% correct. For instance, the word b^ω is not accepted by the second one, but ab^ω is. More generally, if the infinite word stabilizes at some point on b 's, the

in Figure 3.10c has no accepting states, and consequently it accepts now words. In case of the finite-state automata, the two different sets of accepting states led to two different languages, and since the two languages were disjoint, the intersection was empty, which corresponds to the product automaton with no accepting states.

However, the accepting states for Büchi automata are interpreted differently: it's not about being in a state that accepts a word (which would be a finite word), but about hitting a state infinitely often. Now (as made clear in this example), there can be an infinite path that hits an accepting state of the first automaton as well as one accepting state of the second automaton (and thus being accepted), without that it ever hits the 2 accepting states **simultaneously**. So it means the intersection of two infinite-word languages is not captured by the “conjunction” of the accepting states.

Now, the standard “conjunction” construction does not work for Büchi acceptance. Doing a “disjunctive” construction does also not work, i.e., making all states q_{11}, q_{22}, q_{33} of the resulting machine accepting is also not correct (which should be intuitively clear).

What then? Thinking about it, it should be obvious: The automaton from Figure 3.10c combines both constituent automata, capturing their common, simultaneous behavior. We want to capture the joint acceptance, i.e., accept infinite words that are accepted by both automata. The way to do that is not to intersect both accepting sets of states, or build the union; both $F_1 \cap F_2$ and $F_1 \cup F_2$ doesn't work. To capture intersection of the languages, we need to require that both F_1 **and** F_2 have to be visited infinitely many times, it's only that Büchi automata only have one set of accepting states. But if two sets of sets of accepting states, we just need a more general definition of Büchi-automata. This just to the definition of, well, **generalized Büchi automata**. The acceptance condition for standard BAs says, that at least one particular state in the accepting states must be visited infinitely often. Actually, one can alternatively also say that the set F is hit infinitely often. The automata are finite, so if F is visited infinitely often, there must at least be one state in F that is visited infinitely often.

In our example the set of set of accepting states of the product GBA would be

$$\{\{q_{11}\}, \{q_{22}, q_{33}\}\} \quad (3.17)$$

So, we introduced GBAs as **product** of (standard) BAs. Of course, the example is a bit trivial as both automata have the same structure and are deterministic. More generally if one had 2 BAs with accepting sets F_1 and F_2 , the accepting condition for the product is

$$\mathcal{F} = \{F_1 \times Q_2\} \cup \{Q_1 \times F_2\} . \quad (3.18)$$

That means, the set from equation (3.17) is not really correct. For instance one should also include other combinations, namely

$$\{\{q_{11}, q_{12}, q_{13}\}, \dots\} \quad (3.19)$$

number of a 's in the prefix before stabilization is different. That is connected to the fact that the intersection for the finite word languages is empty. Stabilization on b would mean, both automata are spinning in a “self-loop” with b , without any a 's preceding that spinning, but only the first automaton can do that and accept b^ω . Consequently, the intersection does not contain words with finitely many a 's (i.e., words where b stabilizes). Infinitely many b s are allowed in the intersection.

but the states left out from equation (3.17) as they are not reachable (and they are left out from the picture from Figure 3.10c, as well).

Let's apply that insight also to the **specific** situation used for LTL model checking (the case we have already solved). There, the BA results from a transition system. It's a special BA insofar that all states are accepting. In this case, the construction from equation (3.18) degenerates to

$$\mathcal{F} = \{Q_1 \times Q_2\} \cup \{Q_1 \times F_2\} \quad (3.20)$$

(where Q_1 is the BA for the model, not for LTL). As before, it's a set of two sets (of pairs). The first set $Q_1 \times Q_2$ imposes **no restriction** on the accepted words. Now, the other one imposes a restriction, namely the one from LTL. Note that the construction is a **union** of sets, but it's interpreted as **conjunction**, due to the acceptance condition for GBAs: all participating sets need to be visited infinitely often.

This leads to the synchronous product from Definition 3.4.2. It's called there “special case” for the reason mentioned. It should be noted that in the restricted setting, where one of the automata has only accepting states, this Definition of synchronous product works for standard FSAs and for Büchi automata, where actually the step to GBAs is be avoided, as the $Q_1 \times Q_2$ part of equation 3.20 can be ignored and left out, so we have only one set of accepting states, and we don't need GBAs.

To finish the discussion, let's nail down the definition of synchronous product **generalized** BAs. The only thing that needs to be taken care of is the definition of the acceptance condition, and it's a straightforward generalization of the one from equation (3.18):

$$\mathcal{F} = \{F_1 \times Q_2 \mid F_1 \in \mathcal{F}_1\} \cup \{Q_1 \times F_2 \mid F_2 \in \mathcal{F}_2\} \quad (3.21)$$

□

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 3.4.5 adds stuttering to the resulting Büchi-automaton (not the transition system).

Definition 3.4.5 (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.

3.5 Translation LTL to generalized Büchi-automata

The translation we show or at least sketch here is from LTL to Generalized Büchi automata (GBA), not to the plain version of Büchi automata.

Before we do the construction, let's have a look at a few LTL examples and how they can be represented by Büchi automata.

Example 3.5.1. We have seen the example already, under the name *recurrence*. The infinite occurrence of a property, here *green*, is captured by the formula $\Box\Diamond green$. The corresponding and straightforward Büchi automaton is given in Figure 3.11. The exam-

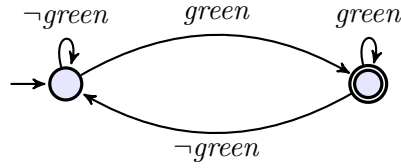


Figure 3.11: Infinitely often green

ple is a good illustration on the Büchi-acceptance: the infinite occurrences of *green* is directly connected to the infinitely many visits to the accepting state of the automaton. prototypical $\square$

Example 3.5.2 (Response). Also this kind of property we have seen already, in Example 3.1.9. Response can be captured in LTL as:

$$\Box(request \rightarrow \Diamond response)$$

Let's abbreviate it to $\Box(a \rightarrow \Diamond b)$. The Büchi-automaton accepting that response property is given in Figure 3.12. $\square$

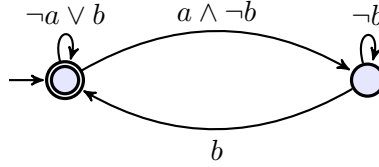


Figure 3.12: Response

Example 3.5.3 (Stabilization). Stabilization (or persistence) can be captured in LTL as:

$$\Diamond\Box a$$

The Büchi-automaton accepting that response property is given in Figure 3.13. $\square$

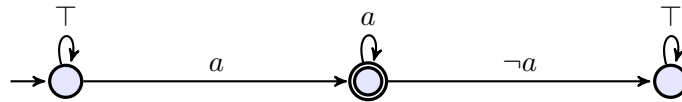


Figure 3.13: Stabilization

The last example captures the dual property from Example 3.5.1. The Büchi automaton (perhaps) feels also slightly different from the others. You can pause to reflect what's different before reading on.

How does the stabilization automaton works anyway? It starts at the initial state, it tries to watch out when the a stabilizes, i.e., to detect the point when that happens. Initially, there's a don't-care self-loop, marked by $\top$, which means it can be always taken. If an a is detected, the automaton can move on to the middle state, and if from that point on, there are only a 's, the word is accepted. If it so happens that, after proceeding to the middle accepting state, $\neg a$ occurs, it is forced to proceed to the third and non-accepting state and stay there forever, thereby rejecting the word. Note, the previous examples never got stuck in this form in non-accepting states, there was always the hope for an edge back to an accepting state.

The rejecting behavior from above, moving in into the middle state and then moving out at the next occurrence of a $\neg a$, does not mean that word being scanned violates the property.

It's just that in its initial state, the automaton has misjudged the situation: seeing an a , instead of staying in the initial state, the automaton took the a as a sign of the beginning of the stabilized sequence of a . Later it may turn out that this was premature, and staying longer in the initial state will lead to acceptance (in case the sequence satisfies the stabilization property). Only it's not knowable how long to wait until exiting the initial state. The definition of acceptance does not care about that: a word is accepted if there **exists** an accepting run, but it does not promise that, given an infinite word, there is exactly one run.

So, one **crucial** difference distinguishing this Büchi-automaton from the two other ones is that it is **non-deterministic**. The situation is worse. For that particular LTL-property and for this particular non-deterministic Büchi-automaton, there is no equivalent deterministic one. We will not prove that, but the discussion around the example may give a feeling that there's not hope for deterministic acceptance for stabilization.

Basic idea of the translation

Here, we don't give construction yet, but some general remarks and some "insightful" property.

The translation translates a minimal version of LTL, i.e., containing only $\bigcirc$ and U as only temporal operators. The most tricky part in the construction, not surprisingly, is how to deal with the until-operator. Until-operators can encode properties like recurrence and stabilizations just discussed. Such properties need properly arranged accepting states, either to be infinitely visited or avoided forever. For each until occurring in the formula to translated, one needs such an arrangement. The translation is easier, if one does not work with plain Büchi automata with one set of accepting states. It's easier to use generalized Büchi-automata, so one can do one set of accepting states *per until*. And then in a later step, one can translate that automaton to a standard Büchi automaton. GBAs were defined in Definition 3.3.11.

Let's refer to the generalized Büchi-automaton for a formula φ by $\mathcal{G}_\varphi$. To get a grip in the construction, we need also get a **mental picture**: what are the states of the automaton, what do they represent, and how are they connected by transitions; we deal with acceptance later.

One key idea in this construction is to connect the syntax with semantics, so to build the model “using” formulas. For LTL, it means the states “consist” of **sets of formulas**. That is a very general idea, and actually, to use syntactic material like sets of formulas to build a semantic model can also be found elsewhere. For instance, completeness of proof systems like for first-order logic can be established that way.

But, as said, the idea is pretty general and thus vague, so what does that mean for us? What we want is a construction such that the states consists of all formulas that hold in that state and the transitions or edges between the states lead from one state to the successors that reflects the changes in the set of formulas that hold. That will basically reflect the interpretation of the $\bigcirc$ -operator.

We said, the states represent *all* formulas that hold in that state. We should qualify that a bit, it's about all *relevant* formulas. If we construct the automaton for $\mathcal{G}_\varphi$, then only formulas that have something to do with φ need to be considered, which will be φ , all its sub-formulas and negations thereof.

Let AP be the (finite) set of propositional variables of interest, i.e., those mentioned in φ . The *alphabet* Σ of the Büchi-automaton then is *sets* of such propositional atoms, i.e.,

$$\Sigma = 2^{AP} .$$

Let's write $A, A', A_i \in \Sigma$ etc. for such sets. Let's look at an infinite sequence $\sigma = A_0 A_1 A_2 \dots$ such that $\sigma \in \llbracket \varphi \rrbracket$ resp.

$$\sigma \models \varphi . \tag{3.22}$$

We commented earlier that the terminology concerning states, paths etc. shifts a bit (and we commented also that this should be unproblematic.) We called the models of LTL formulas *paths* (and used the symbol π for it, where, to add perhaps to the confusion, the pieces of the paths were called states (see Definition 3.1.2). Here, the sequence from equation (3.22) is of the appropriate form.

The infinite sequence σ corresponds to a word accepted by the automaton $\mathcal{G}_\varphi$ we want to construct. As such it corresponds to a word following the edges of the automaton, labelled by Σ . However, it also corresponds to a sequence of states of the automaton: the states are built up from sets of (relevant) formulas that are supposed to hold in that state, and that included propositional atoms.

Now, how about the states of $\mathcal{G}_\varphi$ then? We said that states are sets of formulas, and the automaton is constructed in such a way that the formulas in the states are those which hold in $\mathcal{G}_\varphi$. And that's not just propositional atoms, but also compound formulas, in particular, temporal formulas. That means, the sequence $\sigma = A_0 A_1 A_2 \dots$ of (3.22) cannot be not directly the sequence of states. It corresponds to that state sequence only as far as the propositional atoms are concerned.

Given the word A_i from above, let's consider an infinite sequence of B_i so that each B_i extends the corresponding A_i (by subformulas of φ). I.e., $B_i \supseteq A_i$. Let's refer to that extension of σ by $\hat{\sigma} = B_0 B_1 B_2 \dots$

ms Error: Example: Extension of states

The intention for that sets B_i , ultimately the states of the automaton is that they contain exactly the *language* of that state. That is strong requirement. It also implies that for each formula ψ , either ψ is contained in the state or not. That's a key ingredient of the mention principle to build the model, here the states, out of the syntax, sets of formulas.

The language of a state is meant in the standard way: all the infinite words that are accepted when starting at that state as initial state. With that in mind, we can state the following about the sequence from equation 3.22, connecting the automaton-states and their formulas with the rest of the sequence

$$\psi \in B_i \quad \text{iff} \quad \underbrace{A_i, A_{i+1} A_{i+2} \dots}_{\sigma^i} \models \psi \quad (3.23)$$

The suffix-path σ^i , which satisfies ψ exactly when it is mentioned in B_i (and additionally the intention is that it is a word in the language of the state B_i). The $\hat{\sigma}$, a state sequence, corresponds to an accepting run in $\mathcal{G}_\varphi$

The pool of relevant formulas: the closure of φ As said, the states of $\mathcal{G}_\varphi$ are built from sets of formulas. However, it's not all possible combinations of all possible formulas. There are restrictions on the sets of formulas. One of the restrictions is that one focuses on formulas that can conceivably play a role in an automaton checking φ . Given a formula φ , the pool of formulas taken into account is φ and all its subformulas and negations thereof. Since there are only finitely many formulas in that pool, it results in a **finite-state** automaton. We write $\text{closure}(\varphi)$ for that set.

Definition 3.5.4 (Closure). The *closure* of an LTL formula φ is the set $\text{closure}(\varphi)$ consisting of all subformulas ψ of φ and their negation $\neg\psi$ (where ψ and $\neg\neg\psi$ are identified).

Meaningful sets of formulas, only: maximal consistency. Considering elements from the closure of φ is only one restriction. Not all elements from $2^{\text{closure}(\varphi)}$ make sense considering the purpose of the states formed by sets of formulas. As said, the sets $B \in 2^{\text{closure}(\varphi)}$, i.e., the states of the automaton, are intended to contain exactly all the formulas from the closure that are in the “state-language” of that state, as illustrated in equation (3.23), which involves words with elements B_i from $2^{\text{closure}(\varphi)}$. Of course, the official semantics of automata speaks about words with letters from $\Sigma = 2^{AP}$ following the edges, not words over $2^{\text{closure}(\varphi)}$ but the construction couples them tightly together, for instance in that $B_i \supseteq A_i$ (among other things).

Coming back to the restriction: there is no use of meaningless sets of formulas as states. Literally meaning-less in that they have no interpretation or semantics. For propositional

ms
Error:
Add the
mate-
rial
from
the
slide

atoms, for instance, it makes no sense to have a state B which contains both p and $\neg p$, which would be a state with the intended interpretation that both p and its negation holds there. Such contradictions need to be avoided not just for propositional variables, but for all formulas. So it's required for states B that they don't contain both ψ and $\neg\psi$ at the same time. That assures that for each state, there exists a variable assignment for the propositional variables that make the set B true, at least the propositional, non-temporal part. Such a compositionally non-contradictory set is called **consistent** (one could also say satisfiable).

Is that all that is required for meaningful and useful states? Not yet. We have restricted ourselves to sets of formulas from $\text{closure}(\varphi)$, but we also want that for all such formulas ψ , **either** ψ **or** $\neg\psi$, so that by looking at the state, the logical status of the formula is **determined** by whether the formula is mentioned in the state, or its negation.

So the sets of formulas constituting the states of $\mathcal{G}_\varphi$ or not just consistent subsets of $\text{closure}(\varphi)$, they are **maximal**: Since in the construction, each subformula of φ is either contained in the state or else $\neg\varphi$, one cannot add any further formula from $\text{closure}(\varphi)$ without violating *consistency*. So, states are **maximally consistent sets of formulas from** $\text{closure}(\varphi)$. The book [2], on which the material here is based, calls those sets *elementary* sets, but I think calling them maximally consistent is preferable.

We have discussed at some length the **states** of the automaton to construct. That's enough to give meaning to the propositional, non-temporal part of the formula. Of course, the interpretation of formulas of the form $\bigcirc\psi$ semantically is not decided locally at one state, it speaks about the next states.⁶ So, if a formula $\bigcirc\psi$ is contained in a state (and thereby intended to hold there) is actually true, depends on whether that state is connected by edges only to states which mention ψ . So, for temporal properties, we have to take into account the edges, resp. construct the automaton by connecting states in such a way that $\bigcirc\varphi$ in a state is properly reflected (which will be done later). So, states so far are constructed for the interpretation of non-temporal and propositional aspects, whereas the edges will (additionally) take care of the temporal aspects.

The interpretation of $\bigcirc\psi$ is purely non-local, so there is no condition on states concerning ψ formulas, except the propositional ones discussed, namely that already propositionally, that exactly ψ or else $\neg\psi$ must be included in the state.

But there is another temporal operator, the **until**? As temporal operator, it has no local aspects to take care of in the construction of the states, or has it? The until-operator is the trickiest to translate and actually, there is a **local** aspect of until, and it is the following: A formula $\varphi U \psi$ holds in the **current** state if ψ holds (and that's independent of any edges). That also means, the set of formula can contain **local contradictions** involving until, for instance if it contains both ψ and $\neg(\varphi U \psi)$. This “local” aspect of the temporal operator U leads to the condition of Definition 3.5.7 below.

⁶We introduced it as speaking about *the* next state of paths, but here we are talking about a state in a transition system, so there may be multiple paths starting from a given state, and then $\bigcirc\psi$ refers to multiple next states.

Definition 3.5.5 (Propositional consistency). A set B is *consistent* wrt. propositional logic if

1. $\psi \in B$ implies $\neg\psi \notin B$.
2. $\perp \notin B$.

Definition 3.5.6 (Closed wrt. propositional entailment). A set B is *closed under propositional entailment* if

1. $\varphi_1 \wedge \varphi_2 \in B$ iff $\varphi_1 \in B$ and $\varphi_2 \in B$.
2. if $\top \in \text{closure}(\varphi)$ then $\top \in B$

Definition 3.5.7 (Local consequences of until). A set B is *closed under local entailments* wrt. the *until* operator if

1. $\varphi_2 \in B$ implies $\varphi_1 \ U \ \varphi_2 \in B$
2. $\varphi_1 \ U \ \varphi_2 \in B$ and $\varphi_2 \notin B$ implies $\varphi_1 \in B$.

Definition 3.5.8 (Maximality). A set is *maximal* (relative to the closure of φ), if for all $\psi \in \text{closure}(\varphi)$

$$\psi \notin B \quad \text{implies} \quad \neg\psi \in B .$$

The following definitions just combines the previous one into one:

Definition 3.5.9 (Maximal consistency). Given a LTL-formula φ . A set $B \subseteq \text{closure}(\varphi)$ is *maximally consistent* (or elementary) wrt. φ if it is propositionally consistent, closed under propositional entailment and locally entailed formulas wrt. until, and if it is maximal.

After all this explanations and definitions, let's illustrate it by an example. The example illustrates the concept of closure, also of maximal sets of formulas, in particular maximally consistent extensions as the B_i 's extending the A_i 's in equation (3.23).

Example 3.5.10 ($\varphi = a \ U \ b$: states). Let's construct the GBA for the formula $\varphi = a \ U \ b$. Let's start by constructing the states. We start by defining the closure:

$$\text{closure}(\varphi) = \{a, b, \neg a, \neg b, a \ U \ b, \neg(a \ U \ b)\} \quad (3.24)$$

We soon will list all the maximal the maximally consistent sets from the closure of φ , as those will be the states of the GBA, but let's first also look at a number of states that do **not** qualify to be maximally consistent for various reasons, and therefore are not considered to be part of the automaton.

$$\begin{array}{lll} \{a, b\} & \{a, \neg a, a \ U \ b\} & \\ \{a, b, a \wedge b\} & \{a, b, \neg(a \ U \ b)\} & \{\neg a, \neg b, a \ U \ b\} \end{array} \quad (3.25)$$

$\{a, b\}$ is does not qualify, as the local closure condition requires that if b is part of the set, the $a \ U \ b$ has to be included as well. The second set $\{a, \neg a, a \ U \ b\}$ is propositionally inconsistent. The third set does not qualify, as it includes $a \wedge b$ (and it has the same

problem than the first set). In principle $a \wedge b$ could be considered, as it's required to be closed under logical consequences. However, since the conjunction $a \wedge b$ is not part of the closure, it is not used in the construction. The last formula $\{a, b, \neg(a \cup b)\}$ does not qualify, as it's required that if b is contained in the set, then $a \cup b$ has to be contained as well (but it's negation is added). The last set $\{\neg a, \neg b, a \cup b\}$ is not acceptable as state for a similar reason: the state contains $a \cup b$ and both $\neg b$ nor $\neg b$: But in order to “make sense”, a state that contains $a \cup b$ and $\neg b$, must contain a .

Anyway, the following is the set of all maximal consistent sets, and thus will be the **states** of the automaton:

$$\begin{array}{lll} \{a, b, a \cup b\} & \{a, \neg b, a \cup b\} & \{a, \neg b, \neg(a \cup b)\} \\ \{\neg a, b, a \cup b\} & \{\neg a, \neg b, \neg(a \cup b)\} & \end{array} \quad (3.26)$$

Example 3.5.11.

$$\varphi = a \cup (\neg a \wedge b) \quad (3.27)$$

The closure of that formula is

$$\{a, b, \neg a, \neg b, \neg(\neg a \wedge b), \neg a \wedge b, \varphi, \neg\varphi\} . \quad (3.28)$$

The following combinations are maximally consistent

$$\begin{array}{ll} C_0 &= \{ a, b, \neg(\neg a \wedge b), \varphi \} \\ C_1 &= \{ a, b, \neg(\neg a \wedge b), \neg\varphi \} \\ C_2 &= \{ a, \neg b, \neg(\neg a \wedge b), \varphi \} \\ C_3 &= \{ a, \neg b, \neg(\neg a \wedge b), \neg\varphi \} \\ C_4 &= \{ \neg a, \neg b, \neg(\neg a \wedge b), \neg\varphi \} \\ C_5 &= \{ \neg a, b, \neg a \wedge b, \varphi \} \end{array}$$

The first two columns cover all 4 consistent combinations of involving a and b , respectively their negation. Per combination, the third column is fixed: having chosen the truth values for a and b fixes the truth value of $\neg a \wedge b$, i.e., whether $\neg a \wedge b$ is added or its negation $\neg(\neg a \wedge b)$. The last column is for φ , a temporal formula mentioning until. There, based on the conditions from Definition 3.5.7, a choice for a and b may determine whether φ is added or $\neg\varphi$. This is the case for B_4 and B_5 .

For B_4 , since $\neg(\neg a \wedge b)$ holds and additionally $\neg a$

Example 3.5.12. Let us the formula $\varphi = a \cup (\neg a \wedge b)$ again and its closure from equation (3.27).

Let's look at a small satisfying path or run for that until formula. In the transition system, we assume that a holds in the initial state, b holds in the second. Let also a hold in the second state, though the semantics of until does not insist on that. At that point the execution already satisfies the formula, but for the sake of illustrating the concepts here, let's assume a third state where b holds.

This is shown in the following sequence:

$$\sigma = \{a\}\{a, b\}\{b\} \dots = A_0 A_1 A_2 \dots \quad (3.29)$$

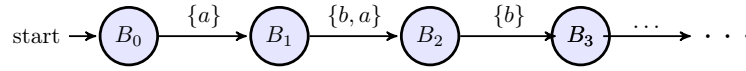


Figure 3.14: Run of the Büchi automaton

The sets A_i are originally, in the transition systems, the propositions that are supposed to hold in the states of the transition system. In the Büchi-automaton, it refers to a word accepted by the automaton via following the edge labels (see 3.14).

The states $B_0, B_1 \dots$ mentioned in the run are states of the Büchi-automaton and thus maximally consistent sets, as explained. Additionally, as indicated in equation (3.23), the $B_i \supseteq A_i$, i.e., each B_i is a maximally consistent extension of the corresponding A_i . For example for B_0 , one could have:

$$B_0 = \{a, \neg b, \neg(\neg a \wedge b), \varphi\} \quad (3.30)$$

B_0 must contain a , it cannot contain $\neg a$, and for the other subformulas, $\neg a \wedge b$ is added in negated form, and φ as is. $\square$

The construction of GBA: transitions and accepting states

We have clarified what the states of the automaton to construct will be, namely maximal consistent sets of formulas from the closure of the given formula. Still missing is the actually core of the construction, namely how to connect the states by transitions, and the definition of the accepting states. The latter is in particular subtle as that will take care of the most complex aspect of the logic, the treatment of the until-formulas.⁷ So we are given a set of propositional variable AP and a formula φ (over those propositional variables), and we construct $\mathcal{G}_\varphi$ such that

$$\llbracket \mathcal{G}_\varphi \rrbracket_{\mathcal{L}} = \llbracket \varphi \rrbracket. \quad (3.31)$$

We have discussed that we take as states maximal consistent subsets of the closure of the given formula. The intention is that a state (= a set of formulas) contains all formulas that will hold in that state, once the construction is finished. If we succeed in doing that, also the initial states are clear, namely all states where φ is contained.

The alphabet of the automaton is fixed, by the selection of propositional variables: the alphabet is all subsets of AP . Note that the states (by the way we have constructed them) carry all kinds of formulas (from the closure), including propositions p and as well as their negation $\neg p$. Of course not both, since we consider only consistent sets, and due to maximality, each state contains either p or else $\neg p$. The alphabet of the automaton are subsets of AP , i.e., the labels carry information over propositional variables, only. And indeed, it will carry only “positive” information over propositional variables, i.e. only p will potentially show up, not $\neg p$. The negation of p is represented in the label by the **absence** of p .

⁷Remember that we are focusing on a restricted, but complete subset of LTL, where U is the only temporal operator besides $\bigcirc$.

That fits with the definition of **transition systems**, where the states were labelled by sets of propositional variables, with the intention that all variables mentioned in the labelling of a state are those which hold true at that state (and the ones not mentioned are false at that point). Now, at the construction of the automaton, we don't have an official state-labelling function that labels the states by set of formulas, but in some way, that what we are doing by the construction of the state, and the intention connect to it, namely that a state contains a formula exactly when it holds in that state (see equation (3.32) below).

That results in the following definition (with not everything yet at its place):

Definition 3.5.13 ($\mathcal{G}_\varphi$ (1)). Given AP and φ , we define $\mathcal{G}_\varphi = (Q, I, 2^{AP}, \mathcal{F}, \delta)$ where

- Q is the set of all *maximal consistent sets* of formulas $B \subseteq \text{closure}(\varphi)$
- $I = \{B \in Q \mid \varphi \in B\}$
- δ and $\mathcal{F}$ as defined in Definitions 3.5.14 and 3.5.15 below.

What remains to be done is the actual core of the construction. We know how the states “look like”, namely sets of formulas. By restricting the attention on sets of subformulas of φ (and their negation), we achieved that we have a finite set of states. We also know the intention of the construction namely that for all states $B \subseteq \text{closure}(\varphi)$,

$$\varphi \in B \quad \text{iff} \quad B \models \varphi \quad (3.32)$$

Transition relation

Next the construction of the transition relation. One can distinguish two aspects. On the one hand it has to deal with the **propositional part**. One key to the construction is to think of the states as places where formulas hold. While that's a good guiding light, and it fits the mental pictures from equations (??) and (3.23): there, a run of the automaton to be constructed passes through an infinite sequence of **states**. However, technically, the language of automata, including Büchi-automata, is traditionally words over symbols on the **transitions** on the machine. So, one needs to **label** the transitions properly. We mentioned earlier that automata and transition systems are quite similar in spirit, transition systems (or Kripke-structures) have state-labels,⁸ and automata have edge-labels. We covered how one can shift from one view to the other, when we constructed the Büchi-automaton of a transition system (see Definition 3.3.16). Basically, we do the same here. We have constructed states “containing” formulas. They are not officially labelled by sets of formulas by some state-labelling function, we could do that actually, but it was done as a mental picture. What we want, however, is to work with automata and thus edge-labelling, not state-labelling, and therefore we use an analogous idea: formulas attached to the state will be included as labels on *outgoing edges*. Unlike what we did in Definition 3.3.16, translating transition systems representing the model to Büchi-automata, here the edge-labels on the automaton are not just single propositions, but sets of propositions, namely all propositions that are mentioned in the start state

⁸Our transition systems have additionally edge-labels which play no role in the construction here.

of the transition we construct.⁹ See Definition 3.5.14(1) and 3.5.14(2). Remember that states B_i in the constructions are maximally consistent sets from the closure of the formula of interest, and “consist” not just of propositions, but also of their negations, and other formulas. See for instance the states from Example 3.5.10. In the outgoing transitions, all the propositions contained in the state form the label. Propositions that occur negated as in $\neg a$ are *left out*, so being absent in the transition label of the automaton represents the negation of the proposition to hold. Remember also that the construction required, that for each formula in the closure, including propositions a , either a is contained or $\neg a$. That explains the use of

$$B \cap AP$$

for constructing labels on transitions that **start** in state B . That covers the propositional “part” of the construction.

What is not explained yet is to which state such a transition **goes to** and that is the second aspect of the construction, which handles the non-propositional, i.e., **temporal properties**.

There are two operators to cover. For the **next operator** $\bigcirc$, it’s fairly obvious: if a state contains $\bigcirc\psi$, then the transitions need to go to those state where ψ is contained. See case 2a in the definition below.

The **until-operator** is quite more subtle, and it will involve also the definition of the accepting states. The until-operator is covered in case 2b. The general intention (not just for until) of the construction is that states that contain a particular formula represents a situation where the formula holds (when the automaton runs). An until formula $\varphi_1 U \varphi_2$ means that eventually φ_2 will hold, and until then, always φ_1 , for all possible runs. A state containing $\varphi_1 U \varphi_2$ imposes the requirement, if φ_2 is not already assumed to hold in the current state, i.e., if $\varphi_2 \notin B$, then we need insist that φ_1 holds and add transitions to states B' which contain $\varphi_1 U \varphi_2$. That is the condition captured in the last line of case 2b, which must hold should $\varphi_2 \notin B$. In a state where $\varphi_1 U \varphi_2$ and $\varphi_2 \in B$, the until-formula actually is true independent of what happens in the future. Consequently, from a state matching this condition, the construction adds transitions **to all states**, because from now on it does not matter where the automaton runs, so it does not matter to which state it transitions.

⁹Actually, it’s not a real difference: *one* transition carrying multiple propositional variables is the same as multiple transitions each labeled by a single propositional variable.

Definition 3.5.14 (Transition relation). The **transition relation**

$$\delta : Q \times 2^{AP} \rightarrow 2^Q \quad (3.33)$$

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

$$\bigcirc\psi \in B \quad \text{iff} \quad \psi \in B' \quad (3.34)$$

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

$$\varphi_1 \, U \, \varphi_2 \in B \quad \text{iff} \quad \begin{array}{l} \varphi_2 \in B \\ (\varphi_1 \in B \quad \text{and} \quad \varphi_1 \, U \, \varphi_2 \in B') \end{array} \quad \text{or} \quad (3.35)$$

The until-construction from above is certainly the most complicated aspect, and actually we have not yet captured it fully in Definition 3.5.14. One important aspect of the transitions for until-formulas $\varphi_1 \, U \, \varphi_2$ is the condition that if that formula is contained in a state, and if φ_2 is not already contained, then φ_1 must be contained, and we need a transition to a state B' where $\varphi_1 \, U \, \varphi_2$ is contained. Thereby allowing transitions, where φ_2 can become true later (and enforcing φ_1 to be true all the way until then). One aspect of until is not yet enforced, though. So far, we covered transitions, so the next step, which is a local condition: If φ_2 is true now, it's fine, if φ_2 is not true, then φ_1 must hold, and there must be transitions to states, where $\varphi_1 \, U \, \varphi_2$ is true. However, the until has a **non-local** requirement, namely that **eventually** φ_2 becomes true. This is a non-local requirement insofar it refers to behavior consisting of multiple transitions (“eventually”). So it's nothing one can fix or enforce by defining the *individual* transitions of the automaton. The construction itself would allow behavior where φ_1 is always true, but never φ_2 (which would be ok for the weak-until, but not for the “real” until).

The accepting states The missing piece of the puzzle is dealt with by **accepting only those runs where φ_2 eventually holds**. That's a non-local condition, and it's the task of the **accepting sets** to assure that. Each until-formula from the closure is represented by its own accepting set. That's another reason for using generalized Büchi automata over standard ones.¹⁰

Definition 3.5.15 (Accepting states).

$$F_{\varphi_1 \, U \, \varphi_2} = \{B \in Q \mid \varphi_2 \in B \text{ or } \varphi_1 \, U \, \varphi_2 \notin B\} . \quad (3.36)$$

Remark 3.5.16 (How to understand the construction, in particular $\mathcal{F}$). The book [2] contains a proof that the Büchi-automaton does what it is supposed to do. The proof is rather technical and omitted in the script and the lecture. Still it may be useful to think about intuition, why the construction works. The most complex part of the whole

¹⁰Another motivation was that intersection of languages is more straightforward.

construction (and where it gets considerably harder than for regular expressions and standard finite state automata) is the treatment of until. So it may be instructive to reflect on why it works, and in particular, which role the accepting states play. Remember that the accepting states are connected only with the U -operations, so it's an important piece of the puzzle.

The construction the Büchi-automaton forms state from sets of formulas (under certain conditions, namely being maximal consistent given the closure of the LTL formula that is used for the problem). The intention is, that if a formula is “contained” in a state, then it holds there. For instance, if $\bigcirc p$ is a formula inside a state, the construction assures that there are only transitions to states which contain p . That's the intention. In the following discussion, we assume that this is more than just intention, we assume that this actually is the case (i.e., the construction works as intended), without having actually established it by doing the actual proof. So, we use words like φ holds in this and this state B , but what we should rather say is that the formula is a member of the set that is a state in the construction, i.e., $\varphi \in B$. But we use sometimes the more intuitive formulation φ holds in B nonetheless.

Another fact we use in the argumentation without explicitly mentioning it repeatedly is that we exploit the notion of *maximality* from Definition 3.5.8: if $\varphi \notin B$, then $\neg\varphi \in B$.

Now we start the argumentation, that a state containing $\varphi_1 U \varphi_2$ has only runs that actually satisfies that formula. That this is so is a consequence of the construction of the transitions (given in equation 3.35 for until) and the choice of the final states (given in equation (3.36)). To argue for it requires to establish two directions: 1) if a state contains $\varphi_1 U \varphi_2$, then all runs starting from there satisfies that formula. That direction should be clear. But we need to establish also the opposite: 2) if a state does not contain that formula, in other words, if a state contains $\neg(\varphi_1 U \varphi_2)$, then all runs starting from there satisfy the negated formula, i.e., no run starting from there satisfies $\varphi_1 U \varphi_2$. It is kind of noteworthy that only the argument for the first direction will actually make use of the final states!

1. So assume that $\varphi_1 U \varphi_2$ is contained in a state, and consider an infinite run that starts from it. Then the definition of transitions from equation (3.34) assures at least one of two situations. If there holds¹¹ φ_2 in the current state already, then the run satisfies the property. If not, then it's guaranteed that φ_1 holds now, and that after one transition, the until formula is contained in that successor state (i.e., still holds), and the **argument repeats itself**.

What still needs to be excluded is that this repeats itself **infinitely often**, i.e., that φ_1 and $\varphi_1 U \varphi_2$ hold permanently. In other words, the argument so far established that the construction of the transitions assure that $\varphi_1 U \varphi_2$ guarantees *weak* until, but does not assume that φ_2 eventually holds. That is the task of the accepting states, see equation (3.36). Alternatively we could formulate the accepting states as implication:

$$F_{\varphi_1 U \varphi_2} = \{B \in Q \mid \varphi_1 U \varphi_2 \in B \text{ implies } \varphi_2 \in B\}. \quad (3.37)$$

¹¹This is one of the points where we could have said that φ_2 is element of the set, not φ_2 holds. It's in the spirit of an inductive argument: we try to establish that the property holds for $\varphi_1 U \varphi_2$, namely that being contained in a set corresponds to that the formula holds, and in the argumentation, we can assume that this already has been established for the subformulas φ_1 and φ_2 .

As explained, the construction of transition assumes that, as long as φ_2 does not hold, $\varphi_1 \ U \ \varphi_2$ will continue to hold. Assume for a **contradiction** that there exists such a run, where φ_1 holds permanently, but never φ_2 (which violates the until-property). For a run of that form, the acceptance condition from (3.36) or from (3.37) degenerates to requiring that φ_2 holds infinitely often. That is a contradiction to the assumption that φ_2 never holds, φ_2 must hold eventually.

2. For the second direction, assume that $\varphi_1 \ U \ \varphi_2$ does not hold, which means $\neg(\varphi_1 \ U \ \varphi_2)$ is contained in a state. For more clarity in this situation, we reformulate the condition for the transition relation and the for the acceptance states contrapositively. The contra-position of condition (3.35) is

$$\neg(\varphi_1 \ U \ \varphi_2) \in B \quad \text{iff} \quad \neg\varphi_2 \in B \quad \text{and} \quad (3.38) \\ (\varphi_1 \in B \quad \text{implies} \quad \neg(\varphi_1 \ U \ \varphi_2) \in B')$$

We have to prove that all runs starting from a state that contains $\neg(\varphi_1 \ U \ \varphi_2)$ violate the until-formula. Such violations can be of one of two possible forms: a “**finite**” violation or an “**infinite**” violation: A finite violation is if $\neg\varphi_1$ occurs before φ_2 ever becomes true. The other form of violation is if there is an infinite sequence of φ_1 . The formulation from equation (3.38) makes clear that the construction of the transitions is done in a way, that also $\neg(\varphi_1 \ U \ \varphi_2)$ is kept (conditionally) invariant, namely as long as φ_1 holds (and φ_1 can hold forever). If φ_1 indeed holds forever, also $\neg\varphi_2$ holds forever, and then that satisfies $\neg(\varphi_1 \ U \ \varphi_2)$, as claimed. That’s the situation we called “infinite violation”). Alternatively, it could be that φ_1 does not hold forever, i.e., there comes a time where $\neg\varphi_1$ is part of a state. Up until that point, the construction guarantees that $\neg\varphi_2$ holds (that’s the “finite violation”). Also that situation is one that violates the until formula.

□

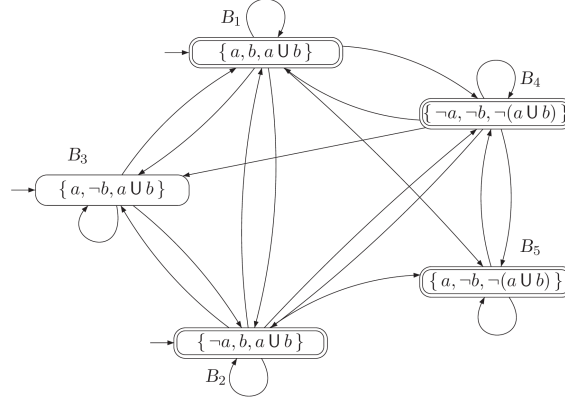
Example 3.5.17 ($\varphi = a \ U \ b$ (cont’d)). Let’s continue with Example 3.5.10, where equation (3.26) lists the maximal consistent sets from the closure of the formula, i.e., the states of the automaton. Next we want to build the **transitions** (and afterwards we determine the **accepting states**).

The transitions are built from $\bigcirc$ -formulas and from U -formulas. The $\bigcirc$ -construction we can ignore in the example, as φ does not contain the $\bigcirc$ -operator.

So let’s focus of U and connect the 5 states according to Definition 3.5.14. The resulting automaton is shown in Figure 3.15.

It could be remarked that the automaton has quite a lot of transitions. That is not directly a consequence of the example or the chosen formula, it’s a consequence of the definition, it typically leads to a lot of transitions (here for until). We are focusing $a \ U \ b$, and each state contains either $a \ U \ b$ or $\neg(a \ U \ b)$ (as the sets are maximal). The intention is (as mentioned) that if a state contains $a \ U$, the formula holds in this state, and we need to construct the transitions in such a way that this actually is the case.

The two temporal operators are responsible for the transitions. The satisfaction of $\bigcirc$ -operator, (which is not part of the example), as **no local aspects**. Whether or no $\bigcirc\varphi$

Figure 3.15: GBA for $\varphi = a U b$

holds in a state exclusively depends on whether φ holds in all successor states or not (and the construction from equation (3.34) makes sure that it fits. The until-operator, in contrast, has **also local aspects**. That was already relevant in the definition of the local consequences of until (Definition 3.5.7): The conditions there (applied to our example) state for instance that if $b \in B$, then $a U b \in B$. The converse implication does not hold: of $a U b \in B$, then both $b \in B$ as well as $\neg b \in B$ are a possibility.

In the first case, namely that both $b \in B$ as well as $a U b \in B$, then $a U b$ is already satisfied at the state B . So $a U b$ is **locally** true, independent of where the transitions go to that state. Therefore, the transitions go to **all other states**. That situation corresponds to the first line of equation (3.35), which does not impose any restrictions on B' . In the automaton, the situation applies to states B_1 and B_2 .

In the second case, namely that $a U b \in B$ but not b , i.e., $\neg b \in B$, it's **not locally already guaranteed** that $a U b$ holds. This is the case, where we cannot transition to all states, but only to those which lead to satisfaction later. That corresponds to line 2 of equation (3.35): locally, $a \in B$, since $\neg b \in B$, and the until-formula requires that until b holds, a must hold. And if we transition to a state, $a U b$ must still hold, i.e., we transition only to states B' where $a U b \in B$. In the example, that gives transitions from B_3 to B_1 and B_2 , but not to B_4 and B_5 .

So we have discussed the situation for $a U b \in B$, and we can do similar argumentation for $\neg(a U b) \in B$, separating situations where $\neg(a U b)$ is already locally satisfied (in which case we transition to all other states) and situations, where this is not the case (in which case we need to transition to a selection of state, that allow the negated formula to be true non-locally). For clarity, let's reformulate the condition contra-positively

$$\neg(a U b) \in B \quad \text{iff} \quad \neg b \in B \quad \text{and} \quad (a \in B \quad \text{implies} \quad \neg(a U b) \in B') \quad (3.39)$$

If $\neg a$ and $\neg b$, it's guaranteed **locally** that $\neg(a U b)$ also holds, independent from any outgoing transitions. Therefore, transitions from such a state goes to all states again. That applies to state B_4 in the example.

However, if $a \in B$, the implication from (3.39) requires to go only to states B' where still $\neg(a \text{ U } b) \in B'$. That applies to state B_5 .

What about the **accepting states**? If one thinks carefully about the above construction, especially the case $a \in B$, $\neg \text{bin}B$, and $a \text{ U } \text{bin}B$, then it says there is a transition to itself; in the picture, B_3 has a self loop. The state captures the situation, where $a \text{ U } b$ is supposed to hold, so eventually b will hold (and until then a), but only not yet right now. So it's legitimate to transition to the same state B_3 where the situation repeats itself. In this way a is constantly true, which is fine, because that's required until we see b . By staying in B_3 we don't give up on eventually seeing b (going to B_1 or B_2), but nothing prevents us for **staying forever** in B_3 . That run accepting $aaaaa \dots$ would violate $a \text{ U } b$. The construction of the transition relation cannot help there, and it's exactly what the **acceptance sets** are good for. In our case there is only one, as we have only one until-operator to consider, and in particular B_3 is not part of it, and that prevents $aaaaa \dots$ from being accepted.

Note that the formula of the example has only one until-operator. Consequently, it has only one accepting set. Technically, it's a *generalized* BA, but Figure 3.15 indicates the accepting states in the way as used from Büchi-automata (and ordinary automata), using doubly-ringed nodes. $\square$

3.6 GBA to BA

We used generalized Büchi-automata as stepping stone in the translation from LTL to Büchi automata. This (not too long) section covers the second part of that translation. What makes GBAs generalized is that they have multiple sets of acceptance states, each of which needs to be visited in an accepted run. Having such multiple acceptance sets is a feature that helped in the translation from LTL to a machine model (where each until-operator is covered by a separate acceptance set).

To show that one can represent a GBA as a BA, one needs to find a way, that captures the combined effect of multiple acceptance sets by a single one. Trivial solutions like building the union of all acceptance sets obviously don't work. But the construction is rather straightforward. The starting point is to realize that the multiple acceptance sets of a GBA express some form of “conjunction”: $\mathcal{F} = \{F_1 \dots F_k\}$ means that F_1 is visited infinitely often, **and** F_2 is visited infinitely often, and so on, for all F_i 's. To achieve this “conjunction” with a single acceptance set exploits the nature of the acceptance conditions, namely of the acceptance sets being visited **infinitely often**. If we take the union of all the F_i 's, then there's no guarantee obviously that all of them are visited infinitely often, when visiting $F_1 \cup F_2 \cup \dots \cup F_k$ infinitely often. But since it's about infinite visitation, we can do it like that: first visit F_1 , then F_2 then F_2 , etc. and finally F_k , and then start again, and do this **cycle of visits** infinitely often. To first visit one of the states of F_1 , then for F_2 etc., one has to work with “**copies**” of the machine. So one arranges a situation, where the structure of the original GBA exists multiple times, the first one representing F_1 as accepting states, the second one F_2 etc. Those sets F_i do not contain accepting states of the standard Büchi automaton we are constructing (with the exception of the states of F_1). Instead they play the following role: if a final state from F_1 in the first copy is reached,

the constructing continues with transitions in the second copy. To continue does not mean there is a transition from a state from F_1 in the first copy to an initial state of the second copy. It simply continues from a state, say q^1 , in F_1 in the first copy to a state with q'^2 in the second copy, i.e., there is a transition $q^1 \xrightarrow{A} q'^2$, if there had been a transition $q \xrightarrow{A} q'$ in the original GBA. And from the final copy, one has to do the analogous transitions back to the first automaton, to close the cycle. From all the accepting sets F_i from the GBA, only F_1 will be taken as the only set of accepting states for the BA.¹² The construction is sketched in Figure 3.16.

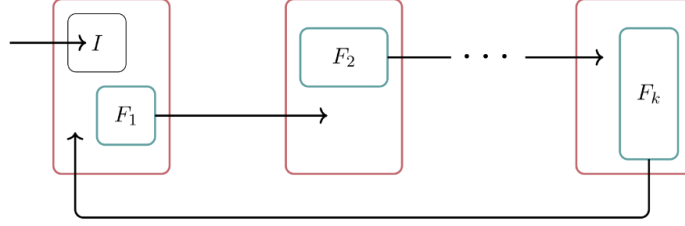


Figure 3.16: GBA to BA: idea

The construction is formalized in Definition 3.6.1.

Definition 3.6.1. Let $\mathcal{G} = (Q, I, \Sigma, \mathcal{F}, \delta)$ with $\mathcal{F} = \{F_1, \dots, F_k\}$ for $k \geq 1$ be an GBA. We define the *equivalent BA* $\mathcal{A} = (Q', I', \Sigma, F', \delta')$ with

- $Q' = Q \times \{1, \dots, k\}$
- $I' = I \times \{1\} = \{\langle q_0, 1 \rangle \mid q_0 \in Q\}$
- $F' = F_1 \times \{1\} = \{\langle q_F, 1 \rangle \mid q_F \in F_1\}$
- $\delta'(\langle q, i \rangle, A) = \begin{cases} \{\langle q', i \rangle \mid q' \in \delta(q, A)\} & \text{if } q \notin F_i \\ \{\langle q', i+1 \rangle \mid q' \in \delta(q, A)\} & \text{if } q \in F_i \text{ and } i < k \\ \{\langle q', 1 \rangle \mid q' \in \delta(q, A)\} & \text{if } q \in F_k \end{cases}$

Example 3.6.2 (From GBA to BA). Let's illustrate the construction on a GBA we have seen before (see Example 3.4.4).

3.7 Checking for emptiness

One final problem to solve is: given an Büchi-automaton, check if its language is empty. For standard automata over finite words, the problem is straightforward: start at the initial state and see if one of the final states is **reachable**. A simple search will do, like depth-first search or some other strategy.

For Büchi-acceptance, the problem gets more complex, but not dramatically so. It involves checking whether some acceptance state can be **reached infinitely often**. Details depend

¹²Actually, any other choice F_i would do.

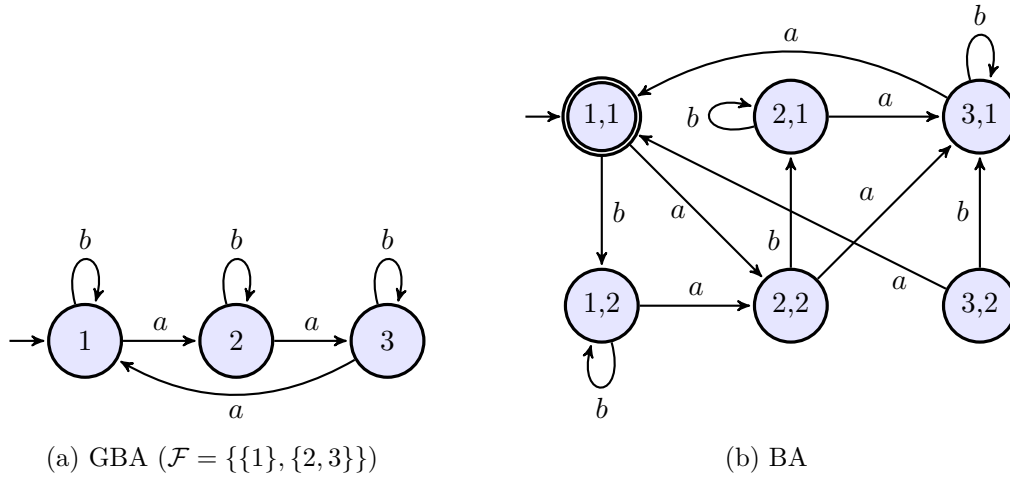


Figure 3.17: From GBA to BA

on whether one deals with plain Büchi-automata or the generalized ones (or some other kinds infinite word automata, which we have not covered).

So, one-time reachability is not enough, one needs to check for some form of *repeated* reachability: from the initial state, reach an accepting state, and then again, and again ... The corresponds to reach an accepting state, and then see if there exist a cycle in the graph starting from that. Because of that shape, it's sometimes illustrated by saying one is looking for a **lasso**.

Example 3.7.1 (Lasso). Let's look at the BA from Figure 3.18. The only accepting state is G . This state is reachable from the initial state A . Additionally, G is part of a cycle, for instance

$$G \rightarrow D \rightarrow H \rightarrow K \rightarrow J \rightarrow G$$

Thus, obvious, gives an infinite run that visits the accepting state G infinitely often and

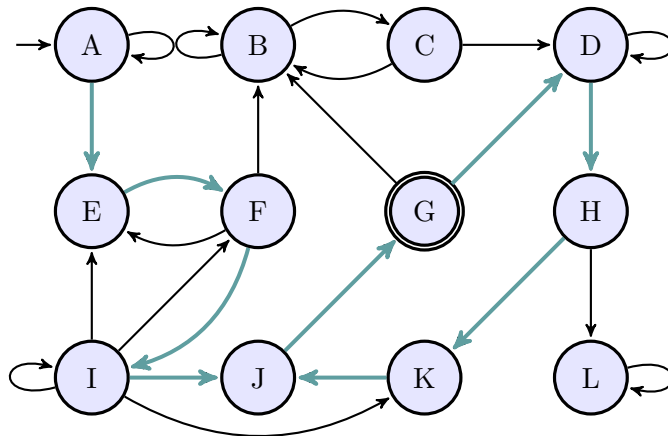


Figure 3.18: BA with reachable "lasso"

consequently, the language of the Büchi automaton is not empty. $\square$

So, to check for emptiness, one could go for a lasso-hunting expedition, searching for reachable lassos. When done properly it would work. But as so often, an initial plausible idea, even if working, leaves room for algorithmic improvement. Here, looking for reachable states that are the starting and end point of a cycle one reachable state after the other will be wasteful. One will re-discover and explore paths through the graph over and over again.

A better idea is based on the notion of **strongly connected components**. It is a concept from graphs and may be known from lectures like “Algorithms and Data Structures”. It’s closely connected to the notion of cycles in a graph, like the one from the previous example. Characteristic for a cycle in a graph is that it consists of nodes, where each node can reach each other.

Definition 3.7.2 (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.

There are efficient algorithms calculating SCCs, for instance one based on doing depth-first search two times, the second time backwards and in a particular order, but we won’t repeat them here.

Example 3.7.3 (SCC). Consider the graph of Figure 3.19. Subsets which are strongly-

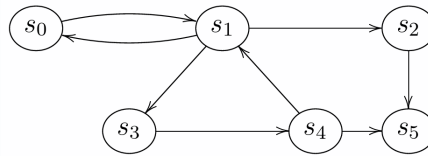


Figure 3.19: Strongly connected component

connected are $S = \{s_0, s_1\}$, $S' = \{s_1, s_3, s_4\}$ and $S'' = \{s_0, s_1, s_3, s_4\}$. From those, only S'' is a strongly connected component. $\square$

A strongly connected components is some form of “cluster of cycles”. As long as one stays within the component, one keeps on cycling in the cluster, if one leaves it, one enters another clusters and there is no way back.

Strongly connected components can be use address the emptiness problem of Büchi-automata. It’s based on the following observation. If one considers an infinite run through a system with finitely many states, it’s unavoidable that some states are visited infinitely often. That is clear.

One can strengthen that observation. Assume an infinite run σ through such a system, there comes point in time in the run, from where on **all subsequent states are visited infinitely often**. That being so, in that run, the mentioned states that visited infinitely

often in a suffix of σ are states which are obviously strongly connected. So, that's the core observation:

given a Büchi automaton $\mathcal{A} = (Q, s_0, \Sigma, \rightarrow, F)$ with accepting run σ , then, there is some *suffix* σ' of σ s.t. every state on σ' is reachable from any other state on σ' . In other words, that set of states is strongly connected and is reachable from an initial state and contains an accepting state.

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

If the language $\llbracket \mathcal{A} \rrbracket_{\mathcal{L}}$ is non-empty, then there is a **counter-example**. The counter-example is one infinite run that, after some point, goes through states that are strongly connected. That would not be an immediately *useful* counter-example. A counter-example is meant as feedback to the user, and a piece of information that is infinitely long is not useful. But it's straightforward to see that should it happen that the infinite counter-example run cannot be finitely represented, there is also a different counter-example run, that can. It just needs to run through the strongly-connected states in an orderly, repetitive fashion (that would be a lasso in the graph/automaton). This “orderly” counter-example 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.

3.8 Expressiveness of LTL and complexity

3.8.1 Expressiveness of LTL and Büchi automata

We have now 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. In contrast, LTL and Büchi-automata 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**.

(Lack of?) expressiveness of LTL

As mentioned, the analogy of Büchi-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-automaton of the following example captures a property that cannot be expressed LTL.

Example 3.8.1. 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:

$$p \wedge \Box(p \rightarrow \bigcirc \neg p) \wedge \Box(\neg p \rightarrow \bigcirc p) \quad (3.40)$$

The formula from equation (3.40) indeed assures that p is false after odd steps, as required, but it's not exactly what was informally given! The LTL formula 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 behavior can be represented by the Büchi-automaton from Figure 3.20a.



Figure 3.20: Büchi automata

The second Büchi-automaton from Figure 3.20b 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:

$$\exists t. t \wedge \Box(t \rightarrow \bigcirc \neg t) \wedge \Box(\neg t \rightarrow \bigcirc t) \wedge \Box(\neg t \rightarrow p) \quad (3.41)$$

The formula, however, is not LTL. It's a formula from what is called $\exists$ LTL (existential LTL). $\square$

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 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 do 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 whether there is 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 3.8.1, in particular the formula from equation

(3.41) hints at a solution for the second approach: It's a result from [16] which states that allowing prefix **existential quantification** over one propositional variable (the t in the example) is enough to repair the mismatch. That version of LTL is sometimes called “existential LTL” or $\exists$ LTL.¹³

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/>.

Comparison of a few formalisms Figure 3.21 compares different well-known temporal logics wrt. their expressivity. Concerning LTL, let's remark that when removing the next-operator $\bigcirc$, the logic becomes strictly less expressive. That fragment is of interest when specifying and verifying properties of **asynchronous** systems. By that we mean systems consisting 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 individual step of a process. Normally some priorities or strategy 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.

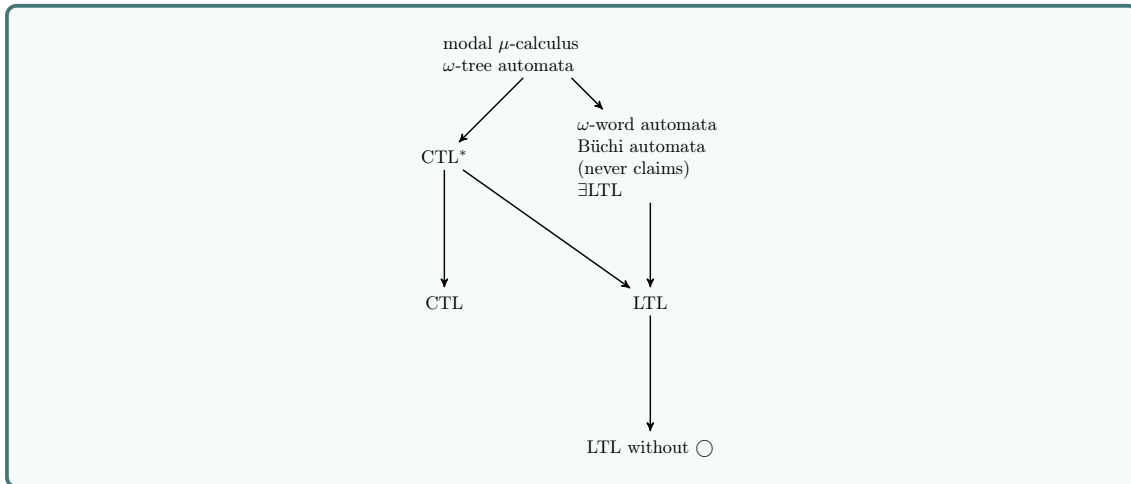


Figure 3.21: Expressivity of a few temporal logics

¹³We will not discuss that variant. Unfortunately there is a *different* formalism with which called existential LTL (or $\exists$ LTL. That's a version where the LTL formula is meant not to hold for all paths, like we do, but is meant to specify that there *exists* a path that satisfies the temporal property. This form of $\exists$ LTL we will encounter later in the context of bounded model checking.

Now, in such an asynchronus picture, and if $\bigcirc$ speaks about transitions of the global system and if $\bigcirc\varphi$ 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 argument here is specifying locally, which is less messy. Locally one could say $\Diamond(x=1)$ instead of $\bigcirc(x=1)$, assuming that the scheduler is **fair** (but otherwise free to choose randomly).

3.8.2 Complexity

The overall construction of automata-based LTL model checking involves a number of constructions. There is the question of complexity of emptiness checking of a BA. Luckily, there are known algorithms that do it in linear time, i.e., they are of complexity $O(S+E)$, where S is the number of states, and E the number of transitions or edges.

But there is another complexity issue, namely the size of the BA.

So then, what’s the size of $\mathcal{G}_\varphi$ given the size of φ ? The number of subformulas of φ is linear in formula length, which means $|closure(\varphi)| \in O(|\varphi|)$. The maximal consistent sets that form the states of the GBA are subsets of $closure(\varphi)$, which means number of states is in $2^{O(|\varphi|)}$. There are at most $|\varphi|$ many acceptance sets, i.e., transforming the GBA to BA results in an automaton with at most $|\varphi| \cdot 2^{O(|\varphi|)} = 2^{O(|\varphi|)}$ many states. Therefore: size of BA (and computation time) is **exponential** in length of formula, and it can be proven, that the exponential growth is unavoidable.

Size of the system The previous considerations showed that “LTL-model checking is exponential”, more precisely that it is exponential in the size of the formula. Other factors are linear, and one could get the impression, thereby negligible. Indeed, the exponential complexity of LTL model checking was an argument sometimes heard against LTL, in particular since there exists similar logics, notably CTL and derivations, which have a linear-time complexity. In the early days of model checking, different logics where explored and different model checker implementations competed for dominance, each year new versions were rolled out with new improvements and techniques refining the existing implementation. Not surprisingly, the exponential complexity of LTL was prominently noted by camps arguing for the superiority of CTL and cousins.

However, the difference exponential-vs-linear complexity is less critical than it sounds. For once, temporal formulas may not be huge. If one tries to specify the temporal behavior of a realistic system, then the specification actually may become long and complex. However, often one specifies many aspects of the system like a response property for than, and a liveness condition for that, and another one, and numerous safety aspects. Seen as a whole, that leads to a large “formula”, but its a conjunction or collection of rather small properties. So one can break it down and check all of the desired properties one after another, instead of building an exponentially large automaton that checks all of the desired property at once. And temporal properties that are genuinely large, like requiring 4 nested until-operators that can’t be separated are seldom (and hard to understand and come up with).

There is another aspect of the exponential-vs-linear complexity discussion. That's the size of the transition system. In the complexity considerations, the size of the transition system factors in as linear, so seemingly negligible compared to the exponential growth of the formula size. However, this ignores that the transition systems can be huge in practice. In particular taking into account an important application area, namely the verification of **concurrent** and **parallel** system. That's systems where *temporal* properties are important, and it's system where testing has weaknesses, and where traditional verification methods are particular hard to apply. Concurrent systems consists on multiple processes or threads or units interacting, and the interaction leads to different possible behaviors, different timings and interleavings, all of which must be explored. The common situation, that the size of a system (or more generally a "problem"), its **state space** grows exponentially with the size of the system is generally known as **state space explosion problem**.¹⁴ For concurrent systems, the state space may grow exponentially with the number of involved processes. State space explosion occurs also in other context, for instance for constraint solving or other techniques, where the state space (to be explore by the algorithms that do the solving) grows exponentially in the number of involved variables. SAT-solving being the most famous example (the prototypical example of an NP-complete problem). Even though the word state space explosion problem is mostly used in the context of model-checking (and the phenomenon is seen as *the* major challenge when applying model checkers), less so in other fields, the growth of a problem description results in an exponential growth of the effort to solve the problem is common in many areas.

¹⁴It's also referred to as state explosion problem sometimes, but that sounds a bit off.

Bibliography

- [1] Andrews, G. R. (2000). *Foundations of Multithreaded, Parallel, and Distributed Programming*. Addison-Wesley.
- [2] Baier, C. and Katoen, J.-P. (2008). *Principles of Model Checking*. MIT Press.
- [3] Baldoni, R., Coppà, E., D’Ella, D. C., Demetrescu, C., and Finocchi, I. (2018). A survey of symbolic execution techniques. *ACM Computing Survey*, 51(3).
- [4] Biere, A. (2009). Bounded model checking. In [5], pages 457–481.
- [5] Biere, A., Heule, M., van Maaren, H., and Walsh, T., editors (2009). *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press.
- [6] Bowen, J. P. and Hinchey, M. G. (1995). Seven more myths of formal methods. *IEEE Software*, 12(3):34–41.
- [7] Bowen, J. P. and Hinchey, M. G. (2005). Ten commandments revisited: a ten-year perspective on the industrial application of formal methods. In *FMICS ’05: Proceedings of the 10th international workshop on Formal methods for industrial critical systems*, pages 8–16, New York, NY, USA. ACM Press.
- [8] Büchi, J. R. (1960). Weak second-order arithmetic and finite automata. *Zeitschrift für mathematische Logik und Grundlagen der Mathematik*, 6:66–92.
- [9] 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.
- [10] Clarke, E., Kroening, D., Ouaknine, J., and Strichman, O. (2005). Computational challenges- in bounded model checking. *Software Tools for Technology Transfer (STTT)*, 7(2):174–184.
- [11] Clarke, E. C., Henzinger, T. A., Veith, H., and Bloem, R., editors (2018). *Handbook of Model Checking*. Springer Verlag.
- [12] Clarke, E. M. (2008). Model checking – my 27-year quest to overcome the state explosion problem. In Cervesato, I., Veith, H., and Voronkov, A., editors, *Logic for Programming, Artificial Intelligence, and Reasoning: 15th International Conference, LPAR 2008, Doha, Qatar, November 22-27, 2008. Proceedings*, Lecture Notes in Artificial Intelligence, pages 182–182. Springer Verlag.
- [13] Clarke, E. M. (2017). SAT-based bounded and unbounded model checking. Available electronically on the net. Data of publication unknown.
- [14] Clarke, E. M., Grumberg, O., and Peled, D. (1999). *Model Checking*. MIT Press.
- [15] Dijkstra, E. W. (1969). Notes on structured programming. Technical Report EWD-249, Technological University, Eindhoven.
- [16] Etesami, K., Wilke, T., and Shuller, R. (2001). Fair simulation relations, parity games, and state space reduction for Büchi automata. In [24], pages 153–167.
- [17] Garfinkel, S. (2005). History’s worst software bugs. Available at <http://archive.wired.com/software/coolapps/news/2005/11/69355?currentPage=all>.

- [18] Godefroid, P., Klarlund, N., and Sen, K. (2005). Dart: Directed automated runtime testing. In [33], pages 213–223.
- [19] Goré, R., Heinle, W., and Heuerding, A. (1997). Relations between propositional normal modal logics: an overview. *Journal of Logic and Computation*, 7(5):649–658.
- [20] Hall, J. A. (1990). Seven myths of formal methods. *IEEE Software*, 7(5):11–19.
- [21] Harel, D., Kozen, D., and Tiuryn, J. (2000). *Dynamic Logic*. Foundations of Computing. MIT Press.
- [22] Holzmann, G. J. (2003). *The Spin Model Checker*. Addison-Wesley.
- [23] Hu, A. J. and Martin, A. K., editors (2014). *Proceedings of the 5th International Conference on Formal Methods in Computer-Aided Design (FMCAD’04), Austin, Texas, USA, November 2004.*, volume 3312 of *Lecture Notes in Computer Science*. Springer Verlag.
- [24] ICALP2001 (2001). *28th Colloquium on Automata, Languages and Programming (ICALP)*, volume 2076 of *Lecture Notes in Computer Science*. Springer Verlag.
- [25] King, J. C. (1976). Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394.
- [26] Lamport, L. (1977). Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, 3(2):125–143.
- [27] Latvala, T., Biere, A., Heljanko, K., and Junttila, T. (2004). Simple bounded ltl model checking. In [23]. An extended version is available as University of Helsinki Technical Report UT-TCS-A92.
- [28] MacKenzie, D. A. (2001). *Mechanizing Proof*. MIT Press.
- [29] Manna, Z. and Pnueli, A. (1992). *The temporal logic of reactive and concurrent systems—Specification*. Springer Verlag, New York.
- [30] McConnell, S. (2004). *Code Complete, Second Edition*. Microsoft Press, Redmond, WA, USA.
- [31] Peled, D. (2001). *Software Reliability Methods*. Springer Verlag.
- [32] Peled, D. (2018). Partial-order reduction. In [11].
- [33] PLDI05 (2005). *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM.
- [34] Pygott, C. (2019). The VIPER microprocessor. In *2019 6th IEEE History of Electrotechnology Conference (HISTELCON)*, pages 14–19.
- [35] Schneider, K. (2004). *Verification of Reactive Systems*. Springer Verlag.
- [36] Shannon, C. E. (1936/1940). A symbolic analysis of relay and switching circuits. Master’s thesis, MIT Press.

Index

- Σ , 26
- σ (run), 28
- $\models$, 5, 7
- π (path), 5
- $\pi \models \varphi$, 7
- π_k , 6
- π^k , 6
- acceptance, 30
- alphabet, 26
- always (temporal operator), 4
- asynchronous product, 41
- Büchi automaton
 - generalized, 34
- branching time, 3
- Büchi acceptance, 33
- closure, 50
- consistency
 - local, 52
 - propositional, 52
- determinism, 27
- deterministic finite state automaton, 28
- DFA, 26
- discrete time, 3
- duality, 13
- equivalence, 8
- eventually (temporal operator), 4
- finite state automaton, 26, 65
- finite state machine, 26
- finite-state automaton, 26
- interleaving, 41
- invariant, 16
- invariant property, 11
- Kripke structure, 26
- labeled transition system, 37
- language, 30
- lasso, 63
- Leslie Lamport, 16
- linear time, 3
- liveness, 14, 16
- local consistency, 52
- logic
 - temporal, 1
- LTL, 2
 - past, 9
- maximality, 52
- maximally consistent, 52
- minimal syntax, 4
- next (temporal operator), 4
- NFA, 26
- non-determinism, 2
- obligation, 17
- parallel composition, 41
- partial order, 29
- past LTL, 9
- path, 5, 6
- permanence, 12, 14
- product
 - asynchronous, 41
- propositional consistency, 52
- QTL, 3
- recurrence, 17, 47
- regular expression, 65
- release (temporal operator), 4
- response, 11, 47
- response property, 10
- run, 28
- safety, 14, 16
- safety property, 31
- satisfaction relation, 7
- SCC, 64
- semantic equivalence, 8
- Spin, 2, 67
- state, 5
- strongly connected component, 64
- strongly connected component, 64
- stutter, 35
- stutter closure, 46
- stutter extension, 35, 36
- stuttering, 19
- syntactic sugar, 4, 7
- temporal logic, 1
- temporal logics, 2

- time interval, 2
- time, 2
 - branching, 3
 - linear, 3
- time line, 2
- total order, 29
- transition function, 28
- transition relation, 28
- transition system, 26
 - labeled, 37
- true concurrency, 29
- until (temporal operator), 4
- valid, 8
- weak until (temporal operator), 4

List of Corrections

Error: Other code	16
Error: Remove the discussion of Kripke-structures	18
Note: Tikz	24
Error: Add the material from the slide	48