Formal Methods module

Petter Nilsson
pettni@caltech.edu

August 31, 2018

Contents

1 Propositional and Temporal Logics

In this course module we will cover the basics of *formal methods*, a class of methods and algorithms to analyze and control the behavior of systems against *formal specifications*. In this first of three lectures we will introduce the two fundamental problems in formal methods: verification and synthesis, cover traditional propositional logic, and introduce temporal logics. We will conclude with a few examples of formal specifications that are useful in practical problems.

The canonical reference for temporal logics and verification is the book by Baier and Katoen [1], whereas the recent book by Belta, Yordanov, and Gol is a good introduction to synthesis [2].

1.1 Formal Methods: Verification and Synthesis

As illustrated in Figure 1.1, the two fundamental problems in formal methods are *verification* and *synthesis*. In the verification problem, a closed system together with a formal specification are given, and the task is to show that the behavior of the system satisfies the specification, or find a counter-example that violates the specification. In the synthesis problem, an open system together with a formal specification are given, and the objective is to synthesize a controller so that the behaviors of the resulting closed-loop system satisfies the specification.

To develop concrete algorithms for these two problems, the class of system models and formal specifications must be precisely defined. The objective of this lecture is to understand what a formal specification is, and become familiar with a popular temporal logic called Linear Temporal Logic (LTL) that can be used to express specifications. First we will revisit traditional propositional logic that is interpreted over *stationary* propositions, and then show how linear temporal logic, which is interpreted over *dynamic* propositions, extends propositional logic and allows meaningful system behaviors to be specified.

Even without conducting any formal verification or synthesis, formal specification can be valuable in its own right. Many engineering companies internally specify "requirements" that systems should satisfy, but as long as those requirements are expressed in a language without mathematically precise semantics (such as English), the requirements are by nature ambiguous and may be interpreted differently by different people. This is part of the reason why the automotive safety standard ISO26262 recommends formal treatment of specifications, even if no actual formal verification is performed[1].

[1]The term used in the standard is "semi-formal verification", which means that a formal specification is evaluated with for instance simulations. As opposed to true formal verification, a finite number of simulations can not capture all possible initial conditions and inputs, and may thus miss corner cases.

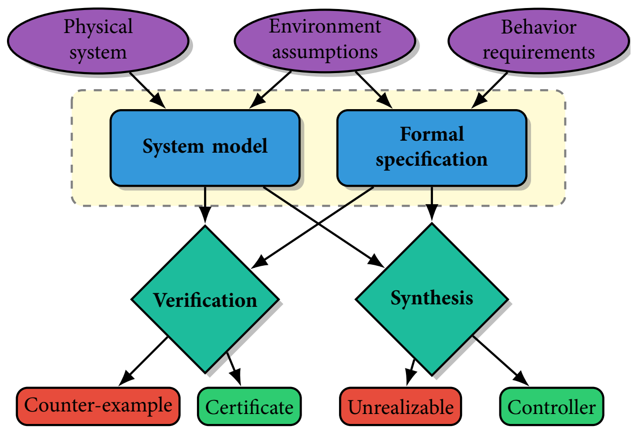

Figure 1.1: The two fundamental problems in formal methods are verification and synthesis. In verification, the behavior of a system with respect to a formal specification is analyzed, whereas in synthesis the objective is to construct a controller that *shapes* the behavior in accordance with a formal specification.

1.2 Propositional Logic: Grammar and Semantics

Propositional logic is a system for constructing *formulas* (or propositions), and is defined by rules for how a formula can be formed (the *grammar*) and rules that dictate how a formula is interpreted (the *semantics*).

1.2.1 Grammar and Semantics

Fundamentally, the grammar is a set of recursive rules that specify how a formula can be composed from other formulas and operators. An *atomic proposition* is a proposition without operators that can either be true ($\top$) or false ($\bot$). Atomic propositions are the fundamental building blocks of a logic. If AP is a set of atomic propositions, then the syntax of propositional logic can be defined using the *Backus-Naur form*.[2]

Definition 1.1 (Propositional logic grammar)**.** *A propositional logic formula φ is formed as follows:*

$$\varphi ::= \top \mid a \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2, \tag{1.1}$$

for $a \in AP$.

These rules say that "a" (an atomic proposition) and "$\top$" (true) as formulas, and that if φ, φ_1, and φ_2 are formulas, then so also $\neg\varphi$ and $\varphi_1 \wedge \varphi_2$ are formulas.

[2]Although not stated in the grammar, we also add parentheses in the formulas to control order of operations.

Example 1.1. *Which of the following expressions are propositional logic formulas?*

$$\top \wedge a, \quad a\neg b, \quad (a \wedge \neg b) \wedge c, \quad ab \wedge \neg\top, \quad \neg(\neg a \wedge \neg b) \wedge \neg\top. \tag{1.2}$$

The grammar (1.1) specifies how formulas are formed, but not what they mean. Defining meaning is the purpose of *semantics*. A definition of semantics complements the grammar by defining exactly when a formula is satisfied. We write $A \models \varphi$ to indicate that a subset $A \subset AP$ of atomic propositions **satisfies** a formula φ.

Definition 1.2 (Propositional logic semantics)**.**

$$\begin{aligned}
&A \models \top && \textit{i.e. any subset of } AP \textit{ satisfies the formula "}\top\textit{"}, \\
&A \models a && \textit{iff } a \in A, \\
&A \models \neg\varphi && \textit{iff } A \not\models \varphi, \\
&A \models \varphi_1 \wedge \varphi_2 && \textit{iff } A \models \varphi_1 \textit{ and } A \models \varphi_2.
\end{aligned} \tag{1.3}$$

Remark 1.1. *The subset $A \subset AP$ of atomic propositions should be thought of as a selection of atomic propositions that evaluate to $\top$ (true), whereas atomic proposition that are not in A evaluate to $\bot$ (false). In the next lecture we will map the output of a dynamical system to a sequence of subsets $A_0A_1A_2\ldots$. Atomic propositions will be associated with subsets of the state space of the system, so that $a \in A_t$ if and only if $\mathbf{x}(t)$ satisfies some condition.*

In the grammar (1.1) we only introduced two operators, $\neg$ (not) and $\wedge$ (and). For notational convenience, the additional derived operators $\bot$ (false), $\implies$ (implication), and $\vee$ (or) are introduced as follows:

$$\begin{aligned}
\bot &:= \neg\top, \\
\varphi_1 \vee \varphi_2 &:= \neg(\neg\varphi_1 \wedge \neg\varphi_2), \\
\varphi_1 \implies \varphi_2 &:= \neg\varphi_1 \vee \varphi_2.
\end{aligned} \tag{1.4}$$

Since these symbols are defined in terms of symbols in (1.1), also the semantics for these symbols follow from Definition 1.2.

A convenient way for understanding formulas of moderate size are *truth tables*. Below is a truth table that illustrates which subsets $A \subset \{a, b\}$ that satisfy the formula $\varphi := a \implies b \equiv \neg a \vee b$.

A	a	b	$\neg a$	$\neg a \vee b$
$\emptyset$	$\bot$	$\bot$	$\top$	$\top$
$\{a\}$	$\top$	$\bot$	$\bot$	$\bot$
$\{b\}$	$\bot$	$\top$	$\top$	$\top$
$\{a, b\}$	$\top$	$\top$	$\bot$	$\top$

Example 1.2. *The formula $\varphi := (a \vee b) \wedge \neg(a \wedge b)$ gives the truth table in Table 1.1. This formula is known as "exclusive or" of a and b: it is true if exactly one of a and b is true.*

Table 1.1: “Exclusive or” truth table.

A	φ
$\emptyset$	$\bot$
$\{a\}$	$\top$
$\{b\}$	$\top$
$\{a, b\}$	$\bot$

1.2.2 Identities

The “and” and “or” operators satisfy following distributive laws

$$\begin{aligned} a \wedge (b \vee c) &\equiv (a \wedge b) \vee (a \wedge c), \\ a \vee (b \wedge c) &\equiv (a \vee b) \wedge (a \vee c), \end{aligned} \tag{1.5}$$

as well as the *de Morgan* laws

$$\begin{aligned} \neg(a \wedge b) &\equiv \neg a \vee \neg b, \\ \neg(a \vee b) &\equiv \neg a \wedge \neg b. \end{aligned} \tag{1.6}$$

Exercise 1.1. *Show that the formula $\varphi := ((\neg a \implies b) \wedge (\neg a \implies \neg b)) \implies a$ is a* tautology, *i.e. that it is satisfied by every assignment $A \subset AP$, by constructing its truth table.*

1.3 Linear Temporal Logic

Now we move from propositional logic that is interpreted over static assignments of truth values, to a logic interpreted over *time-varying* truth assignments. In particular, we introduce Linear Temporal Logic (LTL) which is arguably the most widely used temporal logic.

1.3.1 Grammar and Semantics

Just like propositional logic, LTL is defined in terms of grammar and semantics.

Definition 1.3 (Grammar of LTL)**.** *A linear temporal logic formula φ is formed as follows:*

$$\varphi ::= \top \mid a \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \ \mathbf{U}\ \varphi_2 \mid \bigcirc \varphi. \tag{1.7}$$

for $a \in AP$.

As can be seen from the definition, LTL extends propositional logic with two additional operators $\mathbf{U}$ and $\bigcirc$ called *until* and *next*. Thus, any propositional logic formula is also an LTL formula, but the interpretation of the formula is different. In particular, an LTL formula is evaluated over *sequences* $\sigma = A_0 A_1 A_2 \ldots$ of truth assignments to atomic propositions $A_t \subset AP$. We write $(\sigma, t) \models \varphi$ if the sequence σ satisfies φ at time t, which is defined as follows.

time:	$t=0$	$t=1$	$t=2$	$t=3$	$t=4$ —	$\cdots$
$\bigcirc\varphi$:	$\cdot$ —	φ —	$\cdot$ —	$\cdot$ —	$\cdot$ —	$\cdots$
$\varphi\ \mathbf{U}\ \psi$:	φ —	φ —	φ —	ψ —	$\cdot$ —	$\cdots$
$\Diamond\varphi$:	$\neg\varphi$ —	$\neg\varphi$ —	$\neg\varphi$ —	φ —	$\cdot$ —	$\cdots$
$\Box\varphi$:	φ —	φ —	φ —	φ —	φ —	$\cdots$

Figure 1.2: Illustration of how the LTL operators *next* ($\bigcirc$), *until* ($\mathbf{U}$), *eventually* ($\Diamond$), and *always* ($\Box$) specify temporal behaviors.

Definition 1.4 (Semantics of LTL)**.**

$(\sigma, t) \models \top$	*i.e. any sequence satisfies the formula "$\top$",*
$(\sigma, t) \models a$	*iff $a \in A_t$,*
$(\sigma, t) \models \neg\varphi$	*iff $(\sigma, t) \not\models \varphi$,*
$(\sigma, t) \models \varphi_1 \wedge \varphi_2$	*iff $(\sigma, t) \models \varphi_1$ and $(\sigma, t) \models \varphi_2$,*
$(\sigma, t) \models \bigcirc\varphi$	*iff $(\sigma, t+1) \models \varphi$,*
$(\sigma, t) \models \varphi_1\ \mathbf{U}\ \varphi_2$	*iff there exists $T \geq t$ s.t. $(\sigma, s) \models \varphi_1$ for all $t \leq s < T$, and $(\sigma, T) \models \varphi_2$.*

The symbols that are common with propositional logic have the same interpretation in LTL. However, the symbols $\mathbf{U}$ and $\bigcirc$ enable temporal relationships to be specified.

Example 1.3. *Consider the LTL formulas $\varphi_1 := a\ \mathbf{U}\ b$ and $\varphi_2 = \bigcirc b$, and the truth evaluation sequence*

$$\sigma = \{a\}\{a\}\{a\}\{a,b\}\{a\}\{a\}\{a\}\dots. \tag{1.8}$$

By definition, it holds that $(\sigma, t) \models a$ for all t, but $(\sigma, t) \models b$ only for $t = 3$.

It then follows from Definition 1.4 that $(\sigma, 0) \models \varphi_1$, since the atomic proposition a is satisfied until b becomes satisfied at time 3. However, $(\sigma, t) \not\models \varphi_1$ for any $t \geq 4$, since b never becomes satisfied beyond $t = 3$.

Similarly, $(\sigma, 2) \models \varphi_2$ but $(\sigma, t) \not\models \varphi_2$ for all $t \neq 2$.

In addition to the two operators $\bigcirc$ and $\mathbf{U}$, it is convenient to introduce the *eventually* and *always* operators $\Diamond$ and $\Box$.

$$\Diamond\varphi := \top\ \mathbf{U}\ \varphi, \tag{1.9a}$$
$$\Box\varphi := \neg(\Diamond\neg\varphi). \tag{1.9b}$$

An illustration of the temporal operators is provided in Figure 1.2.

1.3.2 Identities

It follows directly from (1.9b) that the "always" and "eventually" operators are dual to each other in the following sense:

$$\neg\Box\varphi \equiv \Diamond\neg\varphi, \quad \neg\Diamond\varphi \equiv \Box\neg\varphi. \tag{1.10}$$

The interpretation of these formulas is that φ not always being true is equivalent to φ eventually becoming false. Conversely, φ never becoming true is equivalent to it always being false.

The "always" operator is distributive with respect to $\wedge$, and the "eventually" operator is distributive with respect to $\vee$ as capture by the following formulas:

$$\Box(\varphi_1 \wedge \varphi_2) \equiv \Box\varphi_1 \wedge \Box\varphi_2, \quad \Diamond(\varphi_1 \vee \varphi_2) \equiv \Diamond\varphi_1 \vee \Diamond\varphi_2. \tag{1.11}$$

However, the converse is **not true**: the "eventually" operator is not distributive with respect to "and", and "always" is not distributive with respect to "or".

$$\Diamond(\varphi_1 \wedge \varphi_2) \not\equiv \Diamond\varphi_1 \wedge \Diamond\varphi_2, \quad \Box(\varphi_1 \vee \varphi_2) \not\equiv \Box\varphi_1 \vee \Box\varphi_2. \tag{1.12}$$

Example 1.4. *Consider the formulas $\varphi_1 := \Box(a \vee b)$, $\varphi_2 := \Box a \vee \Box b$, and the truth evaluation sequence*

$$\sigma = \{a\}\{b\}\{a\}\{b\}\{a\}\{b\}\dots. \tag{1.13}$$

It holds that $(\sigma, 0) \models \varphi_1$, but $(\sigma, 0) \not\models \varphi_2$, which shows that the second part of (1.12) is indeed a false equivalence.

In addition, the following identities may prove useful when manipulating LTL formulas:

$$\Box\Box\varphi \equiv \Box\varphi, \tag{1.14a}$$
$$\Diamond\Diamond\varphi \equiv \Diamond\varphi, \tag{1.14b}$$
$$\varphi_1 \mathbf{U} \varphi_2 \equiv \varphi_2 \vee (\varphi_1 \wedge \bigcirc(\varphi_1 \mathbf{U} \varphi_2)), \tag{1.14c}$$
$$\Box\varphi \equiv \varphi \wedge \bigcirc\Box\varphi, \tag{1.14d}$$
$$\Diamond\varphi \equiv \varphi \vee \bigcirc\Diamond\varphi. \tag{1.14e}$$

The last three formulas are all examples of dividing the formula into two temporal parts: now ($t = 0$) and the future ($t \geq 1$). For example, $\Box\varphi$ is satisfied if and only if φ is satisfied for $t = 0$ (now) and satisfied for $t \geq 1$ (the future).

Proof of (1.14c). Consider a truth evaluation sequence σ and the specification $\varphi := \varphi_1 \mathbf{U} \varphi_2$. Assume w.l.o.g. that $t = 0$ and consider $(\varphi, 0) \models \varphi$. We can split the condition in Definition 1.4 into two cases: $T = 0$ and $T \geq 1$ so that φ is satisfied if either case is true:

Case $T = 0$: $(\sigma, 0) \models \varphi_2$.

Case $T \geq 1$: $(\sigma, t) \models \varphi_1$ for $0 \leq s < T$ and $(\sigma, T) \models \varphi_2$. However, this is equivalent to $(\sigma, 0) \models \varphi_1$ and the existence of $T' \geq 1$ such that $(\sigma, t) \models \varphi_1$ for $1 \leq s < T'$ and $(\sigma, T') \models \varphi_2$, which can be written $(\sigma, 1) \models \varphi_1 \mathbf{U} \varphi_2$. Using the definition of "next", we can write a formula for this case as $\varphi_1 \wedge \bigcirc(\varphi_1 \mathbf{U} \varphi_2)$. Putting this together with the first case gives the result. □

Proof of (1.14d)-(1.14e). Formula (1.14e) follows from (1.14c) by setting $\varphi_1 = \top$ and recalling (1.9a). Then (1.14d) follows by the substitution $\varphi \to \neg\varphi$, and using the rules (1.10). □

1.4 Specification Examples

To analyze the behavior of a dynamical system we need to define what it means for the output of a system to satisfy a specification. The core idea is to associate atomic propositions with subsets of the system state space via a *labeling* function $L : \mathbb{X} \to 2^{AP}$ so that $a \in L(x)$ if and only if $x \in X_a$ for some $X_a \subset \mathbb{X}$. Then a system trajectory $\mathbf{x}(0)\mathbf{x}(1)\mathbf{x}(2)\ldots$ defines a truth evaluation sequence $L(\mathbf{x}(0))L(\mathbf{x}(1))L(\mathbf{x}(2))\ldots$ of subsets of AP.

The most important specification in practice is arguably *invariance* or *safety* specifications, that are expressed with the temporal operator *always* ($\Box$). If a is an atomic proposition associated with a safe subset $X_a \subset \mathbb{X}$, then the specification $\Box a$ mandates that the system must remain inside of the safe set X_a.

By combining the eventually and always operators, we obtain specification of *eventually-always* type that closely resemble stability specification in traditional control theory. The specification $\Diamond\Box b$ is satisfied if there exists a time T such that $\mathbf{x}(t) \in X_b$ for all $t \geq T$, i.e. the system must reach X_b and remain there for all future time steps.

The same operators combined in different order give specifications of *always-eventually* type, sometimes also known as *recurrence* specifications. If atomic propositions c_1, c_2, and c_3 are associated with subsets X_{c_1}, X_{c_2} and X_{c_3} of a state space $\mathbb{X}$, then the specification $\Box\Diamond c_1 \wedge \Box\Diamond c_2 \wedge \Box\Diamond c_3$ is satisfied if the system visits X_{c_1}, X_{c_2} and X_{c_3} repeatedly over time.

Example 1.5. *Consider a robot with state $\mathbf{x} \in \mathbb{R}^2$, the workspace in Figure 1.3, and the specification*

$$\varphi := \Box a \wedge \bigwedge_{i=1}^{3} \Box\Diamond c_i. \tag{1.15}$$

A truth evaluation sequence that satisfies the specification is

$$(\{a\}\{a\}\{a, c_1\}\{a\}\{a, c_2\}\{a\}\{a, c_3\})^{\omega}, \tag{1.16}$$

where ω signifies infinite repetition. An example trajectory that yields this truth evaluation sequence is marked in Figure 1.3. Note however that (1.16) is not the only sequence that satisfies φ (there are infinitely many), and several trajectories can yield the same truth evaluation sequence.

1.5 Other Temporal Logics

While LTL is the most well known temporal logic, it is by no means the only one. An alternative logic is Computational Tree Logic (CTL), that differs from LTL in that time is branched instead of linear. Concretely, in CTL it is for example possible to express that there should exist a future where a property holds (e.g. "for every execution it is possible to return to the initial state"), even if that future is never realized. CTL and LTL are not directly comparable: there are properties that can be expressed in CTL but not in LTL, and vice versa. There is also a logic called CTL* that is more expressive than both LTL or CTL.

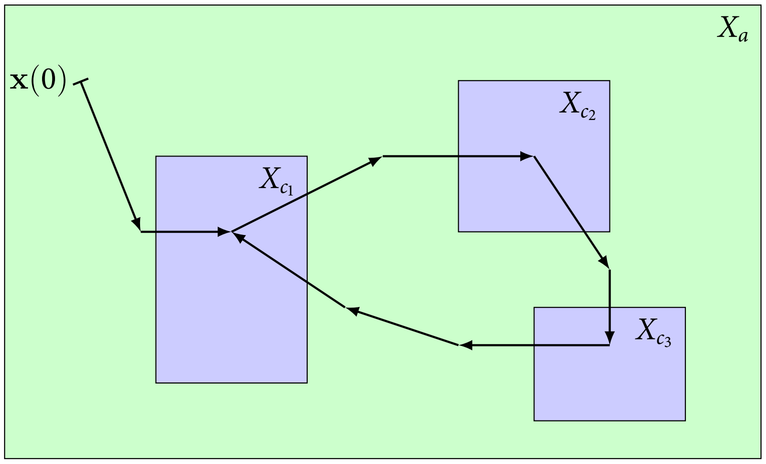

Figure 1.3: Workspace where sets X_a, X_{c_1}, X_{c_2}, X_{c_3} are marked. The atomic proposition a is true at time t if and only if $\mathbf{x}(t) \in X_a$. Thus the marked trajectory results in the truth evaluation in (1.16).

In addition, there are probabilistic extensions of these logics known as Probabilistic LTL (PLTL), Probabilistic CTL (PCTL), and PCTL* that allow for probabilistic quantification of specification satisfaction; Metric Temporal Logic (MTL) and Signal Temporal Logic (STL) amend a finite time interval of enforcement to the temporal operators, and go beyond boolean (true/false) satisfaction and quantify "how much" a specification is satisfied by assigning a real number. There are various methods in the literature for verification and synthesis for these different logics.

References

[1] Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT Press, 2008.

[2] Calin Belta, Boyan Yordanov, and E A Gol. *Formal Methods for Discrete-Time Dynamical Systems*. Springer, 2017.

2 Synthesis for LTL Specifications

Recall that a formal verification or synthesis problem is defined in terms of a system model and a formal specification. In the previous lecture we introduced Linear Temporal Logic (LTL) for expressing specifications. In this lecture we will define a class of system models, and show how LTL synthesis can be conducted algorithmically on such models. The same ideas apply for more general classes of systems as well, but become more difficult to implement due to computational barriers.

2.1 Finite Transition Systems

A basic type of system that for which algorithmic verification and synthesis is relatively straightforward are finite transition systems.

Definition 2.1. *A* ***finite transition system (FTS)*** *is a tuple* $\Sigma = (\mathbb{X}, \mathbb{U}, \longrightarrow)$, *where*

- $\mathbb{X}$ *is a finite set of states* $x \in \mathbb{X}$,
- $\mathbb{U}$ *is a finite set of controlled inputs* $u \in \mathbb{U}$,
- $\longrightarrow \subset \mathbb{X} \times \mathbb{U} \times \mathbb{X}$ *is a transition relation.*

If $(x, u, x') \in \longrightarrow$ *and* $(x, u, x'') \in \longrightarrow$ *implies that* $x' = x''$, *we say that* Σ *is a* ***deterministic finite transition system (DFTS)***.

As an intuitive shorthand notation for transitions, we write $x \xrightarrow{u} x'$ to indicate that $(x, u, x') \in \longrightarrow$, and x' will be called a *u-successor* of x, and x a *u-predecessor* of x'. In a non-deterministic system a state may have several u-successors. A FTS can be viewed as a graph $(\mathbb{X}, \longrightarrow)$, where each edge in $\longrightarrow$ is labeled with an action from $\mathbb{U}$, as shown in Figure 2.1.

2.1.1 Controllers and Trajectories

A controller is a deterministic finite transition system with a fixed initial state whose inputs/outputs match the outputs/inputs of a FTS.

Definition 2.2. *A* ***controller*** *for a transition system* Σ *is a tuple* $C = (\Sigma_C, m_0, \pi)$, *where*

- $\Sigma_C = (\mathbb{M}, \mathbb{X}, \longrightarrow_C)$ *is a transition system with inputs* $x \in \mathbb{X}$ *and states* $m \in \mathbb{M}$,
- m_0 *is the initial state for* Σ_C,

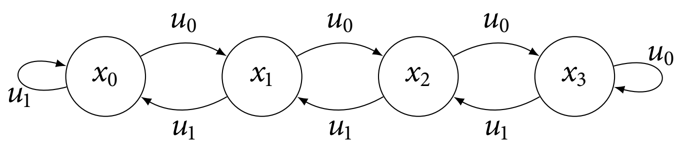

Figure 2.1: Example of a DFTS with $\mathbb{X} = \{x_0, x_1, x_2, x_3\}$, $\mathbb{U} = \{u_0, u_1\}$, and $\longrightarrow = \{(x_0, u_1, x_0), (x_3, u_0, x_3)\} \cup \bigcup_{i=0}^{2} \{(x_i, u_0, x_{i+1}), (x_{i+1}, u_1, x_i)\}$. The system is deterministic since each state-input pair has a unique successor.

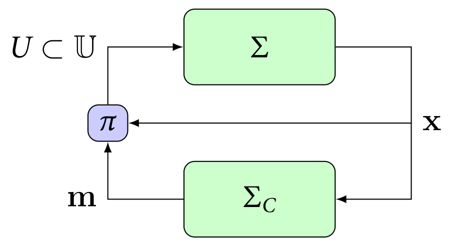

Figure 2.2: Illustration of system-controller interconnection. If the controller is memoryless (i.e. feedback), the transition system Σ_C is not required.

- *$\pi : \mathbb{X} \times \mathbb{M} \to 2^{\mathbb{U}}$ is a function that maps a state of Σ and a state of Σ_C to a set of inputs for Σ.*

*If $\Sigma_C = \Sigma_\emptyset := (\emptyset, \emptyset, \emptyset)$ is the empty transition system, the controller consists of only the mapping $\pi : \mathbb{X} \to 2^{\mathbb{U}}$ and is called a **feedback controller**.*

A controller C can be connected to a system Σ as shown in Figure 2.2. We next define trajectories of controlled systems.

Definition 2.3. *A **trajectory** of a transition system Σ is a sequence of states $\mathbf{x} = \mathbf{x}(0)\mathbf{x}(1)\mathbf{x}(2)\ldots$ with the property that $\mathbf{x}(t) \xrightarrow{\mathbf{u}(t)} \mathbf{x}(t+1)$ for some $\mathbf{u}(t) \in \mathbb{U}$, for all $t \geq 0$. If $\mathbf{u}(t)$ is generated by a controller $C = (\Sigma_C, m_0, \pi)$, i.e. $\mathbf{u}(t) \in \pi(\mathbf{x}(t), \mathbf{m}(t))$, then $\mathbf{x}$ is a C-**controlled trajectory** of Σ.*

Remark that if Σ is deterministic, then each initial state and sequence of inputs corresponds to a unique trajectory. A DFTS is a special case of a Markov decision process where all transition matrices consist exclusively of 0 and 1.

With these definitions in place we can define what it means for a trajectory of the system to satisfy a specification. Consider a labeling map $L : \mathbb{X} \to 2^{AP}$ that assigns a subset of atomic propositions to each state.

Definition 2.4. *A trajectory $\mathbf{x}$ of Σ is said to **satisfy** the specification φ if*

$$(\sigma, 0) \models \varphi, \tag{2.1}$$

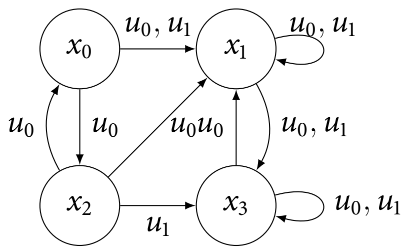

Figure 2.3: A finite transition system Σ for which $\mathrm{Pre}^{\Sigma}(\{x_0, x_1\}) = \{x_0, x_2\}$.

for $\sigma = L(\mathbf{x}(0))L(\mathbf{x}(1))L(\mathbf{x}(2))\ldots$.

*A system Σ and a controller C **satisfy** the specification φ in an initial set $X_0 \subset \mathbb{X}$, written*

$$(\Sigma, C, X_0) \models \varphi, \tag{2.2}$$

if all C-controlled trajectories of Σ satisfy φ.

The problem we address in this lecture is the following.

Problem 2.1. *Given a FTS Σ and an LTL specification φ, synthesize a controller C and a **winning set** $\mathrm{Win}^{\Sigma}(\varphi)$ such that*

$$(\Sigma, C, \mathrm{Win}^{\Sigma}(\varphi)) \models \varphi. \tag{2.3}$$

2.1.2 Solving Reachability Problems on Finite Transition Systems

Later, we will reduce the problem of synthesizing a controller that enforces a specification, to the problem of reaching a final set in an augmented system. Reachability problems are fairly straightforward to solve, the crucial ingredient is the *backwards reachability operator* Pre^{Σ} defined as

$$\mathrm{Pre}^{\Sigma}(X) = \{x : \exists u \in \mathbb{U},\ \forall x' \text{ s.t. } x \xrightarrow{u} x',\ x' \in \mathbb{X}\}. \tag{2.4}$$

The Pre^{Σ} operator returns the set of all initial conditions from where it is possible to control Σ to be in X at the next time step.

Example 2.1. *Consider the transition system in Figure 2.3 and the target set $X_f = \{x_0, x_1\}$. We have that $\mathrm{Pre}^{\Sigma}(X_f) = \{x_0, x_2\}$. For state x_0, action u_1 can be selected which guarantees a transition to the target set X_f, and for state x_2 the action u_0 can be chosen. However, for states x_1 and x_3 there is no way to select an action that guarantees a transition to the target set in the next time step.*

Now the problem of reaching a final set X_f can be solved using Algorithm 1. It iteratively expands the set W with the set of states that can be steered to W, which after convergence will yield the set of all states that can be controlled to X_f. Thus, the returned winning set is the maximal possible winning set. Lines 4-6 in the algorithm store a set of control inputs for each new state, resulting in a (set-valued) feedback controller.

Algorithm 1: Reachability

```
Data: Transition system $\Sigma$, final set $X_f$
Result: Winning set $W$, feedback control map $\pi : \mathbb{X} \to 2^{\mathbb{U}}$, s.t. $(\Sigma, \pi, W) \models \Diamond(x \in X_f)$
1 $W := X_f$;
2 initialize $\pi : \mathbb{X} \to 2^{\mathbb{U}}$;
3 while $\mathrm{Pre}^{\Sigma}(W) \not\subset W$ do
4     for $x \in \mathrm{Pre}^{\Sigma}(W) \setminus W$ do
5         set $\pi(x) := \{u \in \mathbb{U} : \forall x' \text{ s.t. } (x, u, x') \in \longrightarrow, x' \in W\}$;
6     end
7     $W := X_f \cup \mathrm{Pre}^{\Sigma}(W)$;
8 end
```

2.2 Synthesis for scLTL Formulas

An *automaton* is a FTS with some additional structure in the form of initial states and an acceptance condition. Acceptance conditions implicitly define a family of input sequences that result in trajectories that satisfy the acceptance condition, and it turns out that an LTL formula φ can be translated into an automaton, so that the set of input sequences that satisfy the acceptance condition is exactly the set of sequences that satisfy the formula. We will first consider a fragment of LTL with formulas that can be translated into automata with particularly simple acceptance conditions.

Definition 2.5. *A **deterministic finite-state automaton (DFSA)** over a set of atomic propositions AP is a tuple $\mathcal{A} = (\mathbb{Q}, 2^{AP}, \longrightarrow, q_0, Q_f)$ where*

- *$\mathbb{Q}$ is a finite set of automaton states $q \in \mathbb{Q}$,*
- *2^{AP} is a set of inputs $A \in 2^{AP}$,*
- *$\longrightarrow \subset \mathbb{Q} \times 2^{AP} \times \mathbb{Q}$ is a deterministic transition relation,*
- *$q_0 \in \mathbb{Q}$ is an initial state,*
- *$Q_f \subset \mathbb{Q}$ is a set of accepting states.*

A **trajectory** $\mathbf{q} = \mathbf{q}(0)\mathbf{q}(1)\mathbf{q}(2)\ldots$ of $\mathcal{A}$ is defined as for FTSs, with the additional requirement that $\mathbf{q}(0) = q_0$.

Definition 2.6. *Consider an DFSA $\mathcal{A}$ and an input sequence $\boldsymbol{A} = \boldsymbol{A}(0)\boldsymbol{A}(1)\boldsymbol{A}(2)\ldots$; it results in a trajectory $\boldsymbol{q} = \boldsymbol{q}(0)\boldsymbol{q}(1)\boldsymbol{q}(2)\ldots$ of $\mathcal{A}$. We say that $\boldsymbol{A}$ is **accepted** by $\mathcal{A}$ if and only if there exists a $n \in \mathbb{N}$ such that $\boldsymbol{q}(n) \in Q_f$.*

This type of automaton can represent formulas in a fragment of LTL called syntactically co-safe Linear Temporal Logic (scLTL). This fragment is defined by the grammar

$$\varphi ::= \top | a | \neg a | \varphi_1 \wedge \varphi_2 | \varphi_1 \vee \varphi_2 | \bigcirc \varphi | \varphi_1 \mathbf{U} \varphi_2. \tag{2.5}$$

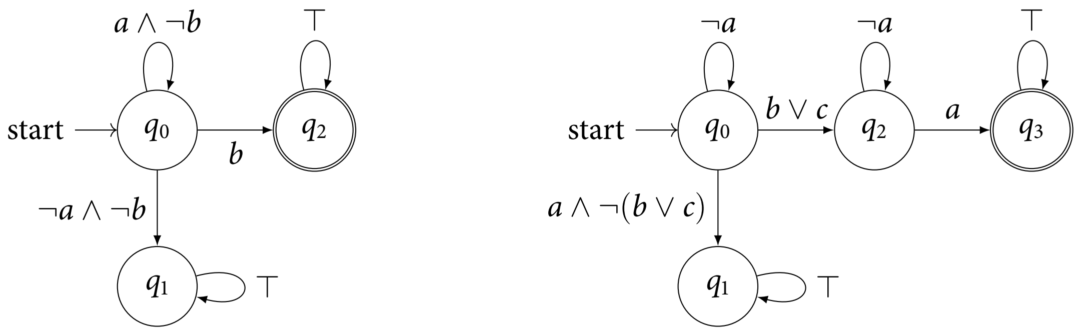

Figure 2.4: DFSAs corresponding to the two formulas $a\ \mathbf{U}\ b$ (left) and $\neg a\ \mathbf{U}\ (b \vee c) \wedge \Diamond a$ (right). The initial states are marked with an arrow, and the accepting states are marked with double circles. In both examples, q_1 is a *trap state* from where it is not possible to reach an accepting state.

The "eventually" operator $\Diamond$ is defined as before. However, since the grammar only allows negation ($\neg$) in front of atomic propositions, the "always" operator can not be expressed in scLTL. In general, a formula φ in scLTL has the property that any infinite sequence $A_0A_1A_2\dots$ has a finite *good prefix* $A_0A_1\dots A_T$ such that all sequences that start with this good prefix satisfy φ.

Remark 2.1. *It may seem impractical to not be able to express safety specifications using the always operator. However, for many robotic tasks it is possible to enforce safety until the specification is "complete" using the "until" operator. For example, a specification $\Box a \wedge \Diamond b$ can be replaced by a specification $a\ \mathbf{U}\ b$.*

A crucial result that connects formulas and automata is now as follows [2, 6].

Theorem 2.1. *For any scLTL specification φ, there exists a DFSA $\mathcal{A}_\varphi$ such that for all sequences $\boldsymbol{A} = \boldsymbol{A}(0)\boldsymbol{A}(1)\boldsymbol{A}(2)\dots$ with $\boldsymbol{A}(t) \in 2^{AP}$,*

$$(\boldsymbol{A}, 0) \models \varphi \quad \textit{if and only if} \quad \mathcal{A}_\varphi \textit{ accepts } \boldsymbol{A}. \tag{2.6}$$

The translation from a scLTL formula to a DFSA can be done with tools such as `scheck`[1]. Figure 2.4 shows DFSA representations of the two formulas $a\ \mathbf{U}\ b$ and $\neg a\ \mathbf{U}\ (b \vee c) \wedge \Diamond a$.

Due to this result, we can reduce the problem of synthesizing a controller that enforces a scLTL specification, to a reachability problem. To this end, we define the *product* $\Sigma \otimes \mathcal{A}$ between a FTS Σ and a DFSA $\mathcal{A}$, which is the interconnection of Σ and $\mathcal{A}$ via a labeling function L that determines the inputs to $\mathcal{A}$ from states in Σ.

Definition 2.7. *Let $\Sigma = (\mathbb{X}, \mathbb{U}, \longrightarrow_\Sigma)$ be a FTS, $L : \mathbb{X} \to 2^{AP}$ a labeling function, and $\mathcal{A} = (\mathbb{Q}, 2^{AP}, \longrightarrow_\mathcal{A}, q_0, Q_f)$ a DFSA. The **product** $\Sigma \otimes \mathcal{A} = (\mathbb{X} \times \mathbb{Q}, \mathbb{U}, \longrightarrow)$ is the FTS with transition relation $\longrightarrow$ defined as follows:*

$$(x, q) \xrightarrow{u} (x', q') \text{ if and only if } x \xrightarrow{u}_\Sigma x' \text{ and } q \xrightarrow{L(x')}_\mathcal{A} q'. \tag{2.7}$$

[1]`http://tcs.legacy.ics.tkk.fi/users/tlatvala/scheck/`

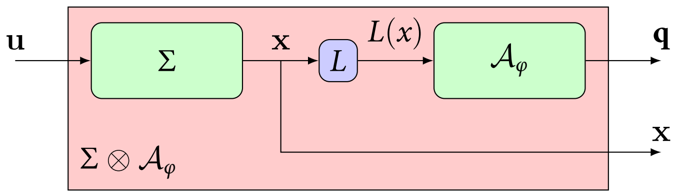

Figure 2.5: Illustration of the product system $\Sigma \otimes \mathcal{A}_\varphi$ that has input u and output (x, q).

The product system is illustrated in Figure 2.5. Using this construction we can now construct an algorithm that solves Problem 2.1 for the case when φ is a scLTL formula.

Algorithm 2: scLTL synthesis

```
   Data: FTS $\Sigma$, scLTL formula $\varphi$
   Result: Controller $C$, winning set $W$ such that $(\Sigma, C, W) \models \varphi$
1  Translate $\varphi$ to a DFSA $\mathcal{A}_\varphi$;
2  Form the product $\Sigma \otimes \mathcal{A}_\varphi$;
3  Synthesize a (feedback) controller $\pi_{\text{Reach}}$ and a winning set $W_{Reach}$ using Algorithm 1 such
     that $(\Sigma \otimes \mathcal{A}_\varphi, \pi_{\text{Reach}}, W_{Reach}) \models \Diamond((\mathbf{x}, \mathbf{q}) \in \mathbb{X} \otimes Q_f)$;
4  Let $W := \{x : (x, q_0) \in W_{Reach}\}$;
5  Refine $\pi_{\text{Reach}}$ to a controller $C$ (with memory) for $\Sigma$;
```

When the feedback reachability controller for $\Sigma \otimes \mathcal{A}_\varphi$ is mapped to a controller for Σ, the internal controller memory will in general contain an internal representation of $\mathcal{A}_\varphi$ since the controller needs to know how much progress that has been made towards satisfying φ (i.e., reaching an accepting state in $\mathcal{A}_\varphi$).

2.3 Synthesis for General LTL Formulas

In the previous section we considered synthesis in the scLTL fragment, which is easy due to the simple acceptance condition for DFSAs. For general formulas, other types of automata are required. Any LTL formula can be translated into a deterministic Rabin automaton (DRA) $\mathcal{R} = (\mathbb{Q}, 2^{AP}, \longrightarrow, q_0, \{(G_i, R_i)\}_{i=1}^{I})$ which has an acceptance condition defined in terms of sets $G_i, R_i \subset \mathbb{Q}$. Using LTL syntax, the Rabin acceptance condition can be written as follows:

$$\psi_{\text{Rab}} := \bigvee_{i=1}^{I} \left(\Diamond\Box\neg(\mathbf{q} \in R_i) \wedge \Box\Diamond(\mathbf{q} \in G_i)\right). \tag{2.8}$$

i.e. for some i, the set R_i is eventually avoided, and the set G_i is reached infinitely often. That is, the Rabin acceptance condition is fulfilled for an input sequence $\mathbf{A}(0)\mathbf{A}(1)\mathbf{A}(2)\ldots$ if it results in a trajectory $\mathbf{q} = \mathbf{q}(0)\mathbf{q}(1)\mathbf{q}(2)\ldots$ of $\mathcal{R}$ such that

$$(\mathbf{q}, 0) \models \psi_{\text{Rab}}. \tag{2.9}$$

A product $\Sigma \otimes \mathcal{R}_\varphi$ between a finite transition system and a Rabin automaton is again a Rabin automaton, so the LTL synthesis problem can be reduced to the problem of synthesizing a controller that enforces ψ_{Rab}. Thus, similarly to the scLTL case we can construct the following synthesis algorithm for general LTL formulas.

Algorithm 3: LTL synthesis

```
    Data: FTS $\Sigma$, LTL formula $\varphi$
    Result: Controller $C$, winning set $W$ such that $(\Sigma, C, W) \models \varphi$
1   Translate $\varphi$ to a DRA $\mathcal{R}_\varphi$;
2   Form the product $\Sigma \otimes \mathcal{R}_\varphi$;
3   Synthesize a controller $\pi_{\text{Rab}}$ and a set $W_{\text{Rab}}$ such that $(\Sigma \otimes \mathcal{R}_\varphi, \pi_{\text{Rab}}, W_{\text{Rab}}) \models \psi_{\text{Rab}}$;
4   Let $W := \{x : (x, q_0) \in W_{\text{Rab}}\}$;
5   Refine $\pi_{\text{Rab}}$ to a controller $C$ for $\Sigma$;
```

The translation into a DRA can be done with the tool `ltl2dstar`[2] and there are known algorithms for solving the Rabin synthesis problem [2, 9].

2.4 Synthesis via Fixed Points

The algorithms above are general in scope, but the translation from specification to automaton can be costly: the worst-case size of the Rabin automaton is $\mathcal{O}(2^{2^{|\varphi| \log |\varphi|}})$, where $|\varphi|$ is the length of the LTL formula. Also the size of a DFSA that represents a scLTL formula can be double-exponential in the length of the formula in the worst case [2].

2.4.1 GR(1) Fragment

Due to these discouraging complexity results, researchers have identified fragments of LTL for which more favorable algorithms are available, and that avoid constructions of product automata by doing fixed-point computations directly on the state space. One example is reachability specifications on the form $\varphi := \Diamond a$: they can be solved using Algorithm 1 without the need to form a product system. A more expressive fragment is the Generalized Reactivity (1) (GR(1)) fragment, which consists of formulas on the form

$$\varphi := \left(a^{\text{init}} \wedge \Box a^{\text{safe}} \wedge \bigwedge_{j=1}^{J} \Box\Diamond a^{\text{rec},j} \right) \implies \left(g^{\text{init}} \wedge \Box g^{\text{safe}} \wedge \bigwedge_{i=1}^{I} \Box\Diamond g^{\text{rec},i} \right). \tag{2.10}$$

[2] http://www.ltl2dstar.de/

The two parts of the formula have natural interpretations as *assumptions* (left of the implication) and *guarantees* (right of the implications). There is a triple-nested fixed-point algorithm with polynomial complexity that can solve the synthesis problem for this type of specification [3]. GR(1) is the fragment used in the synthesis tool TuLiP [4].

2.4.2 Another Fragment

Another fragment for which fixed-point algorithms are available is specifications on one of the following dual forms:

$$\varphi_1 := \Box a \wedge \Diamond\Box b \wedge \bigwedge_{i\in I} \Box\Diamond c^i, \quad \varphi_2 := \Diamond a \vee \Box\Diamond b \vee \bigvee_{i\in I} \Diamond\Box c^i. \tag{2.11}$$

We borrow notation from μ-calculus [5] for succinct expression of fixed points. Let $\kappa : 2^{\mathbb{X}} \longrightarrow 2^{\mathbb{X}}$ be a mapping that is monotone with respect to set inclusion, i.e., $V \subset W \implies \kappa(V) \subset \kappa(W)$. Due to monotonicity, repeated applications of κ necessarily converge due to finiteness of $\mathbb{X}$.

Definition 2.8. *The* ***greatest fixed point of*** κ*, written* $\nu V\ \kappa(V)$*, is the value after convergence of the set sequence*

$$V_0 = \mathbb{X}, \quad V_{k+1} = \kappa(V_k).$$

Correspondingly, the ***smallest fixed point of*** κ*, written* $\mu V\ \kappa(V)$*, is the value after convergence of*

$$V_0 = \emptyset, \quad V_{k+1} = \kappa(V_k).$$

Furthermore, let $\llbracket a \rrbracket = \{x \in \mathbb{X} : a \in L(x)\}$ denote the subset of states that satisfy an atomic proposition. With this notation the winning set computed in Algorithm 1 can be written on the compact form

$$\mathrm{Win}^{\Sigma}(\Diamond a) = \mu V\ \llbracket a \rrbracket \cup \mathrm{Pre}^{\Sigma}(V). \tag{2.12}$$

Also with this notation, the winning sets for specifications on the forms (2.11) can be computed as follows [8]:

$$\mathrm{Win}^{\Sigma}(\varphi_1) = \mu V_2\ \nu V_1 \bigcap_{i\in I} \mu V_0\ \ \mathrm{Pre}^{\Sigma}(V_2) \cup \left(\llbracket b \rrbracket \cap \llbracket c^i \rrbracket \cap \mathrm{Pre}^{\Sigma}(V_1)\right) \cup \left(\llbracket b \rrbracket \cap \mathrm{Pre}^{\Sigma}(V_0)\right), \tag{2.13}$$

$$\mathrm{Win}^{\Sigma}(\varphi_2) = \nu V_2\ \mu V_1 \bigcup_{J\subset I} \nu \begin{bmatrix} \vdots \\ V_0^J \\ \vdots \end{bmatrix} \begin{bmatrix} \vdots \\ \kappa^J(V_2, V_1, \{V_0^K\}) \\ \vdots \end{bmatrix}, \tag{2.14a}$$

$$\kappa^J\left(V_2, V_1, \{V_0^K\}\right) = \begin{pmatrix} \llbracket a \rrbracket \cup \left(\llbracket b \rrbracket \cap \mathrm{Pre}^{\Sigma}(V_2)\right) \cup \mathrm{Pre}^{\Sigma}(V_1) \\ \cup \left(\left(\bigcap_{i\in J} \llbracket c^i \rrbracket\right) \cap \mathrm{Pre}^{\Sigma}\left(\bigcup_{L\subset 2^J} V_0^L\right)\right) \end{pmatrix}. \tag{2.14b}$$

These fixed-point algorithms are implemented in the Matlab toolbox ARCS[3], along with routines that extract controllers that enforce the specifications inside of the winning sets [7].

References

[2] Calin Belta, Boyan Yordanov, and E A Gol. *Formal Methods for Discrete-Time Dynamical Systems*. Springer, 2017.

[3] Roderick Bloem et al. "Synthesis of Reactive(1) designs". In: *Journal of Computer and System Sciences* 78 (2012), pp. 911–938. DOI: 10.1016/j.jcss.2011.08.007.

[4] Ioannis Filippidis et al. "Control design for hybrid systems with TuLiP: The Temporal Logic Planning toolbox". In: *2016 IEEE Conference on Control Applications, CCA 2016* (2016), pp. 1030–1041. DOI: 10.1109/CCA.2016.7587949.

[5] Dexter Kozen. "Results on the propositional μ-calculus". In: *Theoretical Computer Science* 27.3 (1983), pp. 333–354. DOI: 10.1016/0304-3975(82)90125-6.

[6] Orna Kupferman and Moshe Y. Vardi. "Model checking of safety properties". In: *Formal Methods in System Design* 19.3 (2001), pp. 291–314. DOI: 10.1023/A:1011254632723.

[7] Oscar Bulancea Lindvall, Petter Nilsson, and Necmiye Ozay. "Nonuniform abstractions, refinement and controller synthesis with novel BDD encodings". In: *Proceedings of the IFAC Conference on Analysis and Design of Hybrid Systems (to appear)*. July 2018. arXiv: 1804.04280 [cs.SY].

[8] Petter Nilsson, Necmiye Ozay, and Jun Liu. "Augmented finite transition systems as abstractions for control synthesis". In: *Discrete Event Dynamic Systems* 27.2 (2017), pp. 301–340. DOI: 10.1007/s10626-017-0243-z.

[9] Nir Piterman and Amir Pnueli. "Faster Solutions of Rabin and Streett Games". In: *IEEE Symposium on Logic in Computer Science* (2006), pp. 275–284. DOI: 10.1109/LICS.2006.23.

[3] https://github.com/pettni/abstr-refinement

3 Introduction to TuLiP

The purpose of this lecture is to give a short introduction to the TuLiP toolbox. While a large part of TuLiP is built for continuous-state system abstraction, we will focus on the discrete-state synthesis algorithms here. We will show how to define a simple synthesis problem by constructing a finite transition system (FTS) and giving a GR(1) specification, and how the synthesis problem can subsequently be solved.

3.1 Construct a FTS

A FTS is represented by the `tulip.transys.FTS` class. We create a system that represents the non-deterministic transition system in Figure 3.1.

```python
from tulip import transys

fts = transys.FTS()
fts.add_nodes_from(['s0', 's1', 's2', 's3'])
fts.sys_actions.add_from(['u1', 'u2'])

fts.add_edge('s0', 's1', sys_actions='u1')
fts.add_edge('s0', 's0', sys_actions='u2')
fts.add_edge('s0', 's1', sys_actions='u2')

fts.add_edge('s1', 's3', sys_actions='u1')
fts.add_edge('s1', 's1', sys_actions='u2')

```

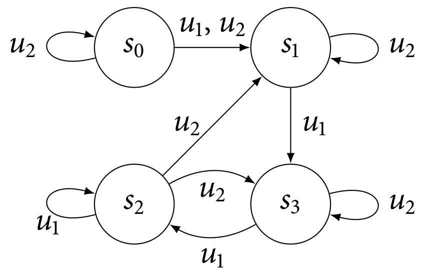
$$\begin{tikzcd}
s_0 \arrow[loop left, "u_2"] \arrow[r, "{u_1, u_2}"] & s_1 \arrow[loop right, "u_2"] \arrow[d, "u_1"] \\
s_2 \arrow[loop left, "u_1"] \arrow[ur, "u_2"] \arrow[r, bend left, "u_2"] & s_3 \arrow[loop right, "u_2"] \arrow[l, bend left, "u_1"]
\end{tikzcd}$$

Figure 3.1: Example non-deterministic FTS.

```python
fts.add_edge('s2', 's2', sys_actions='u1')
fts.add_edge('s2', 's1', sys_actions='u2')
fts.add_edge('s2', 's3', sys_actions='u2')

fts.add_edge('s3', 's2', sys_actions='u1')
fts.add_edge('s3', 's3', sys_actions='u2')

fts.plot() # visualize the system
```

We can also add assumptions about the initial state of the system, specify a set of atomic propositions AP, and define the mapping $L : \mathbb{X} \to 2^{AP}$. Any policy synthesized with TuLiP will be valid only for the given initial states. This is in contrast to the algorithms in Lecture 2 that returned the maximal set of initial states. Here we define s_0 to be the single initial state, add two atomic propositions a and b, and add atomic propositions to two of the states. This corresponds to the labeling function $L(s_2) = \{a\}$, $L(s_3) = \{b\}$, and $L(s_0) = L(s_1) = \emptyset$.

```python
fts.states.initial.add('s0')

fts.atomic_propositions.add_from(['a', 'b'])
fts.node['s2']['ap'] |= {'a'}
fts.node['s3']['ap'] |= {'b'}

fts.plot() # visualize the system
```

3.2 Give a Specification

The next step is to define a specification. TuLiP supports GR(1) specifications that are on the following form via the `tulip.spec.GRSpec` class:

$$\varphi := \left(\varphi_e^0 \wedge \bigwedge_{i \in I_1} \Box \varphi_e^{s_i} \wedge \bigwedge_{i \in I_2} \Box\Diamond \varphi_e^{r_i} \right) \implies \left(\varphi_s^0 \wedge \bigwedge_{i \in I_1} \Box \varphi_s^{s_i} \wedge \bigwedge_{i \in I_2} \Box\Diamond \varphi_s^{r_i} \right). \tag{3.1}$$

The different parts of the formula are given to `GRSpec` via the keywords `env_init` for φ_e^0, `env_safety` for $\varphi_e^{s_i}$, and `env_prog` for $\varphi_e^{r_i}$, and similarly for the system part of the specification. In addition, we can specify additional boolean variables controlled by the environment and the system via the keywords `env_vars` and `sys_vars`. Here we give a specification $\Box\Diamond a \wedge \Box\Diamond b$ that only depends on atomic propositions in the FTS, so we only need to use the `sys_prog` keyword. Values to keywords are given as python `set`s, where each member of the set corresponds to e.g. a boolean specification $\varphi_s^{r_i}$.

```python
from tulip import spec

sys_prog = set()        # empty set
sys_prog |= {'a', 'b'} # set union

formula = spec.GRSpec(sys_prog=sys_prog)

print formula.pretty() # print the formula on readable form
```

3.3 Solve Synthesis Problem

A few additional things need to be specified before solving the problem. It is important to understand the distinction between variables that are controlled by the environment and variables controlled by the system. We can assign the FTS to either belong to the environment, or to the system, where the owner controls resolution of nondeterminism. The way we have treated FTSs until now corresponds to *environment* ownership, where the `sys_actions` of the FTS become system variables, and atomic propositions, transitions, and `env_actions` in the FTS become environment variables that are constrained by the dynamics of the FTS.

In addition, we need to specify how initial conditions are treated. Here we specify `qinit = '\E \A'` which means that we want a strategy such that there exists an initialization of system variables, such that for all possible initializations of environment variables, the controller can control the system variables to comply with the formula. Finally, we set `moore = False` which tells the solver to synthesize a *Mealy* controller which has access to the current state of environment variables. The alternative is to synthesize a *Moore* controller, which has to fix the control action before the environment moves.

Then we can call `tulip.synth.synthesize` where we again specify that the FTS `fts` is owned by the environment.

```python
from tulip import synth

fts.owner = 'env'
formula.qinit = '\E \A'
formula.moore = False

cont = synth.synthesize('omega', formula, env=fts)

cont.plot() # plot the control strategy
```

TuLiP solves the problem by translating the transition system into a GR(1) specification φ_Σ, and feeding the combined specification $\varphi \wedge \varphi_\Sigma$ to an external solver—in this case `'omega'`[1]. A useful

[1]This is different from the procedure in Lecture 2 where we solved a fixed point problem on the state space of a system

command for debugging is to print the combined specification as follows:

```python
print synth._spec_plus_sys(formula, fts, None, False, False).pretty()
```

3.4 Learn More

This example, together with two more advanced examples, are available at `https://github.com/pettni/tulip-examples`. Additional resources for TuLiP:

- The user guide: `https://tulip-control.sourceforge.io/doc/`
- API for the `FTS` class: `https://tulip-control.sourceforge.io/api-doc/tulip.transys.transys.FiniteTransitionSystem-class.html`
- API for the `GRSpec`: class `https://tulip-control.sourceforge.io/api-doc/tulip.spec.form.GRSpec-class.html`
- API for `synth.synthesize()`: `https://tulip-control.sourceforge.io/api-doc/tulip.synth-module.html#synthesize`

and a specification automaton.

List of Acronyms

DFSA	deterministic finite-state automaton.
DFTS	deterministic finite transition system.
DRA	deterministic Rabin automaton.
FTS	finite transition system.
LTL	Linear Temporal Logic.
scLTL	syntactically co-safe Linear Temporal Logic.