Roberto Bruni, Ugo Montanari

Models of Computation

– Monograph –

May 16, 2016

Springer

Mathematical reasoning may be regarded rather schematically as the exercise of a combination of two facilities, which we may call intuition and ingenuity.

Alan Turing[1]

[1] The purpose of ordinal logics (from Systems of Logic Based on Ordinals), Proceedings of the London Mathematical Society, series 2, vol. 45, 1939.

Preface

The origins of this book lie their roots on more than 15 years of teaching a course on formal semantics to graduate Computer Science to students in Pisa, originally called *Fondamenti dell'Informatica: Semantica* (*Foundations of Computer Science: Semantics*) and covering models for imperative, functional and concurrent programming. It later evolved to *Tecniche di Specifica e Dimostrazione* (*Techniques for Specifications and Proofs*) and finally to the currently running *Models of Computation*, where additional material on probabilistic models is included.

The objective of this book, as well as of the above courses, is to present different *models of computation* and their basic *programming paradigms*, together with their mathematical descriptions, both *concrete* and *abstract*. Each model is accompanied by some relevant formal techniques for reasoning on it and for proving some properties.

To this aim, we follow a rigorous approach to the definition of the *syntax*, the *typing* discipline and the *semantics* of the paradigms we present, i.e., the way in which well-formed programs are written, ill-typed programs are discarded and the way in which the meaning of well-typed programs is unambiguously defined, respectively. In doing so, we focus on basic proof techniques and do not address more advanced topics in detail, for which classical references to the literature are given instead.

After the introductory material (Part I), where we fix some notation and present some basic concepts such as term signatures, proof systems with axioms and inference rules, Horn clauses, unification and goal-driven derivations, the book is divided in four main parts (Parts II-V), according to the different styles of the models we consider:

IMP: imperative models, where we apply various incarnations of well-founded induction and introduce λ-notation and concepts like structural recursion, program equivalence, compositionality, completeness and correctness, and also complete partial orders, continuous functions, fixpoint theory;

HOFL: higher-order functional models, where we study the role of type systems, the main concepts from domain theory and the distinction between lazy and eager evaluation;

CCS, π: concurrent, non-deterministic and interactive models, where, starting from operational semantics based on labelled transition systems, we introduce the notions of bisimulation equivalences and observational congruences, and overview some approaches to name mobility, and temporal and modal logics system specifications;

PEPA: probabilistic/stochastic models, where we exploit the theory of Markov chains and of probabilistic reactive and generative systems to address quantitative analysis of, possibly concurrent, systems.

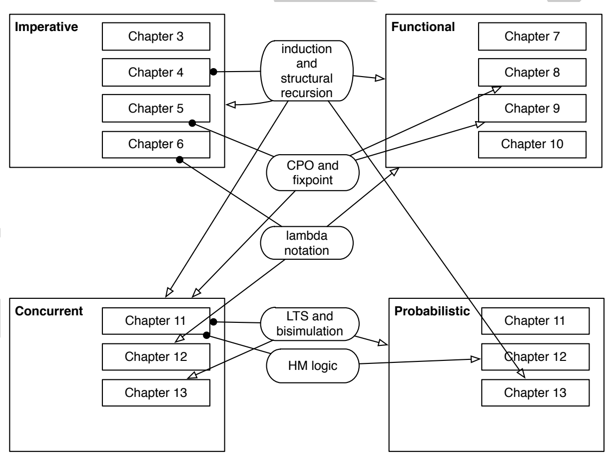
Each of the above models can be studied in separation from the others, but previous parts introduce a body of notions and techniques that are also applied and extended in later parts.

Parts I and II cover the essential, classic topics of a course on formal semantics.

Part III introduces some basic material on process algebraic models and temporal and modal logic for the specification and verification of concurrent and mobile systems. CCS is presented in good detail, while the theory of temporal and modal logic, as well as π-calculus, are just overviewed. The material in Part III can be used in conjunction with other textbooks, e.g., on model checking or π-calculus, in the context of a more advanced course on the formal modelling of distributed systems.

Part IV outlines the modelling of probabilistic and stochastic systems and their quantitative analysis with tools like PEPA. It poses the basis for a more advanced course on quantitative analysis of sequential and interleaving systems.

The diagram that highlights the main dependencies is represented below:



The diagram contains a squared box for each chapter / part and a rounded-corner box for each subject: a line with a filled-circle end joins a subject to the chapter where it is introduced, while a line with an arrow end links a subject to a chapter or part where it is used. In short:

Induction and recursion: various principles of induction and the concept of structural recursion are introduced in Chapter 4 and used extensively in all subsequent chapters.

CPO and fixpoint: the notion of complete partial order and fixpoint computation are first presented in Chapter 5. They provide the basis for defining the denotational semantics of IMP and HOFL. In the case of HOFL, a general theory of product and functional domains is also introduced (Chapter 8). The notion of fixpoint is also used to define a particular form of equivalence for concurrent and probabilistic systems, called bisimilarity, and to define the semantics of modal logic formulas.

Lambda-notation: λ-notation is a useful syntax for managing anonymous functions. It is introduced in Chapter 6 and used extensively in Part III.

LTS and bisimulation: Labelled transition systems are introduced in Chapter 11 to define the operational semantics of CCS in terms of the interactions performed. They are then extended to deal with name mobility in Chapter 13 and with probabilities in Part V. A bisimulation is a relation over the states of an LTS that is closed under the execution of transitions. The before mentioned bisimilarity is the coarsest bisimulation relation. Various forms of bisimulation are studied in Part IV and V.

HM-logic: Hennessy-Milner logic is the logic counterpart of bisimilarity: two state are bisimilar if and only if they satisfy the same set of HM-logic formulas. In the context of probabilistic system, the approach is extended to Larsen-Skou logic in Chapter 15.

Each chapter of the book is concluded by a list of exercises that span over the main techniques introduced in that chapter. Solutions to selected exercises are collected at the end of the book.

Pisa,
February 2016

Roberto Bruni
Ugo Montanari

Acknowledgements

We want to thank our friend and colleague Pierpaolo Degano for encouraging us to prepare this book and submit it to the EATCS monograph series. We thank Ronan Nugent and all the people at Springer for their editorial work. We acknowledge all the students of the course on *Models of Computation (MOD)* in Pisa for helping us to refine the presentation of the material in the book and to eliminate many typos and shortcomings from preliminary versions of this text. Last but not least, we thank Lorenzo Galeotti, Andrea Cimino, Lorenzo Muti, Gianmarco Saba, Marco Stronati, former students of the course on *Models of Computation*, who helped us with the LaTeX preparation of preliminary versions of this book, in the form of lecture notes.

Contents

Acronyms

$\sim$	operational equivalence in IMP (see Definition 3.3)
$\equiv_{den}$	denotational equivalence in HOFL (see Definition 10.4)
$\equiv_{op}$	operational equivalence in HOFL (see Definition 10.3)
$\simeq$	CCS strong bisimilarity (see Definition 11.5)
$\approx$	CCS weak bisimilarity (see Definition 11.16)
$\approxeq$	CCS weak observational congruence (see Section 11.8.2)
$\cong$	CCS dynamic bisimilarity (see Definition 11.18)
$\sim_E$	π-calculus strong early bisimilarity (see Definition 13.3)
$\sim_L$	π-calculus strong late bisimilarity (see Definition 13.4)
$\simeq_E$	π-calculus strong early full bisimilarity (see Section 13.5.3)
$\simeq_L$	π-calculus strong late full bisimilarity (see Section 13.5.3)
$\approx_E$	π-calculus weak early bisimilarity (see Section 13.5.4)
$\approx_L$	π-calculus weak late bisimilarity (see Section 13.5.4)
$\mathscr{A}$	interpretation function for the denotational semantics of IMP arithmetic expressions (see Section 6.2.1)
ack	Ackermann function (see Example 4.18)
Aexp	set of IMP arithmetic expressions (see Chapter 3)
$\mathscr{B}$	interpretation function for the denotational semantics of IMP boolean expressions (see Section 6.2.2)
Bexp	set of IMP boolean expressions (see Chapter 3)
$\mathbb{B}$	set of booleans
$\mathscr{C}$	interpretation function for the denotational semantics of IMP commands (see Section 6.2.3)
CCS	Calculus of Communicating Systems (see Chapter 11)
Com	set of IMP commands (see Chapter 3)
CPO	Complete Partial Order (see Definition 5.11)
$\mathrm{CPO}_\perp$	Complete Partial Order with bottom (see Definition 5.12)
CSP	Communicating Sequential Processes (see Section 16.2)
CTL	Computation Tree Logic (see Section 12.2.2)
CTMC	Continuous Time Markov Chain (see Definition 14.15)
DTMC	Discrete Time Markov Chain (see Definition 14.14)

Env	set of HOFL environments (see Chapter 9)
fix	(least) fixpoint (see Definition 5.2.2)
FIX	(greatest) fixpoint
gcd	greatest common divisor
HML	Hennessy-Milner modal Logic (see Section 11.6)
HM-Logic	Hennessy-Milner modal Logic (see Section 11.6)
HOFL	A Higher-Order Functional Language (see Chapter 7)
IMP	A simple IMPerative language (see Chapter 3)
int	integer type in HOFL (see Definition 7.2)
Loc	set of locations (see Chapter 3)
LTL	Linear Temporal Logic (see Section 12.2.1)
LTS	Labelled Transition System (see Definition 11.2)
lub	least upper bound (see Definition 5.7)
$\mathbb{N}$	set of natural numbers
$\mathscr{P}$	set of closed CCS processes (see Definition 11.1)
PEPA	Performance Evaluation Process Algebra (see Chapter 16)
Pf	set of partial functions on natural numbers (see Example 5.13)
PI	set of partial injective functions on natural numbers (see Problem 5.12)
PO	Partial Order (see Definition 5.1)
PTS	Probabilistic Transition System (see Section 14.4.2)
$\mathbb{R}$	set of real numbers
$\mathscr{T}$	set of HOFL types (see Definition 7.2)
Tf	set of total functions from $\mathbb{N}$ to $\mathbb{N}_\perp$ (see Example 5.14)
Var	set of HOFL variables (see Chapter 7)
$\mathbb{Z}$	set of integers

Part IV
Concurrent Systems

This part focuses on models and logics for concurrent, interactive systems. Chapter 11 defines the syntax, operational semantics and abstract semantics of CCS, a calculus of communicating systems. Chapter 12 introduces several logics for the specification and verification of concurrent systems, namely LTL, CTL and the μ-calculus. Chapter 13 studies the π-calculus, an enhanced version of CCS, where new communication channels can be created dynamically and communicated to other processes.

Chapter 12
Temporal Logic and the μ-Calculus

Formal methods will never have a significant impact until they can be used by people that don't understand them. (Tom Melham)

Abstract As we have briefly discussed in the previous chapter, modal logic is a powerful tool that allows to check important behavioural properties of systems. In Section 11.6 the focus was on Hennessy-Milner logic, whose main limitation is due to its finitary structure: a formula can express properties of states up to a finite number of steps ahead and thus only local properties can be investigated. In this chapter we show some extensions of Hennessy-Milner logic that increase the expressiveness of the formulas by defining properties about finite and infinite computations. The most expressive language that we present is the μ*-calculus*, but we start by introducing some other well-known logics for program verification, called *temporal logics*.

12.1 Specification and Verification

Reactive systems, such as those composed by parallel and distributed processes, are characterised by non-terminating and highly nondeterministic behaviour. Reactive systems have become widespread in our daily activities, from banking to healthcare, and in software-controlled safety critical systems, from railways control systems to space craft control systems. Consequently, gaining maximum confidence about their trustworthiness has become an essential, primary concern. Intensive testing can facilitate the discovery of bugs, but cannot guarantee their absence. Moreover, developing test suites that grant full coverage of possible behaviours is difficult in the case of reactive systems, due to their above mentioned intrinsic features.

Fuelled by impressive, world fame disaster stories of software failures[1] that (maybe) could have been avoided if formal methods would have been employed, over

[1] Top famous stories include the problems with the Therac 25 radiation therapy engine that in the period 1985-1987 caused the death of several patients by releasing massive overdoses of radiation; the floating-point division bug in the Intel Pentium P5 processor due to an incorrectly coded lookup table and discovered in 1994 by Professor Thomas R. Nicely at Lynchburg College; and the launch failure in Ariane 5.01 maiden flight due to an overflow in data conversion that caused a hardware exception and finally led to self-destruction.

the years, formal methods have provided an extremely useful support in the design of reliable reactive systems and in gaining high confidence that their behaviour will be correct. The application of formal logics and model checking is nowadays common practice in the early and advanced stages of software development, especially in the case of safety-critical industrial applications. While disaster stories do not prove, by themselves, that failures could have been avoided, in the last three decades many success stories can be found in several different areas, such as, e.g., that of mobile communications and security protocols, chip manufacturing, air-traffic control systems, nuclear plants emergency systems.

Formal logics serve to write down unambiguous specifications about how a program is supposed to behave and to reason about system correctness. Classically, we can divide the properties to be investigated in three categories:

safety: properties expressing that something bad will not happen;
liveness: properties expressing that something good will happen;
fairness: properties expressing that something good will happen infinitely often.

The first step in extending HM-logic is to introduce the concept of time, which was present only in a primitive form in the modal operators. This will extend the expressiveness of modal logic, making it able to talk about concepts like “at the next instant of time”, “always”,“never” or “sometimes”. When several options are possible, we will also use *path quantifiers*, meaning “for all possible future computations” and “for some possible future computation”. In order to represent the concept of time in our logics we have to model it in some mathematical fashion. In our discussion we assume that the time is discrete and infinite.

We start by introducing temporal logics and then present the μ-calculus, which comes equipped with least and greatest fixpoint operators. Notably, most modal and temporal logics can be defined as fragments of the μ-calculus, which in turn provides an elegant and uniform framework for comparison and system verification. Translations from temporal logics to the μ-calculus are of practical relevance, because not only they allow to re-use algorithms for the verification of μ-calculus formulas to check if temporal logics are satisfied, but also because temporal logic formulas are often more readable than specifications written directly in the μ-calculus.

12.2 Temporal Logic

Temporal logic shares similarities with HM-logic, but:

- temporal logic is based on a set of *atomic propositions* whose validity is associated with a set of states, i.e., the observations are taken on states and not on (actions labelling the) arcs;
- temporal operators allow to look further than the “next” operator of HM-logic;
- as we will see, the choice of representing the time as linear (linear temporal logic) or as a tree (computation tree logic) will lead to different types of logic, that roughly correspond to the trace semantic view vs the bisimulation semantics view.

12.2.1 Linear Temporal Logic

In the case of *Linear Temporal Logic* (LTL) the time is represented as a line. This means that the evolutions of the system are linear, they proceed from a state to another without making any choice. The formulas of LTL are based on a set P of *atomic propositions* p, which can be composed using the classical logic operators together with the following temporal operators:

O: is called *next* operator. The formula $O\phi$ means that ϕ is true in the next state (i.e., in the next instant of time). Some literature uses X or N in place of O.
F: is called *finally* operator. The formula $F\phi$ means that ϕ is true sometime in the future.
G: The formula $G\phi$ means that ϕ is always (*globally*) valid in the future.
U: is called *until* operator. The formula $\phi_0 U \phi_1$ means that ϕ_0 is true until the first time that ϕ_1 is true.

LTL is also called *Propositional Temporal Logic* (PTL).

Definition 12.1 (LTL formulas). The syntax of LTL formulas is defined as follows:

$$\begin{array}{rl}\phi ::= & true \mid false \mid \neg\phi \mid \phi_0 \wedge \phi_1 \mid \phi_0 \vee \phi_1 \mid \\ & p \mid O\,\phi \mid F\,\phi \mid G\,\phi \mid \phi_0\,U\,\phi_1\end{array}$$

where $p \in P$ is any atomic proposition.

In order to represent the state of the system while the time elapses we introduce the following mathematical structure.

Definition 12.2 (Linear structure). A *linear structure* is a pair (S,P), where P is a set of *atomic propositions* and $S : P \to \wp(\mathbb{N})$ is a function assigning to each proposition $p \in P$ the set of time instants in which it is valid; formally:

$$\forall p \in P.\ S(p) = \{n \in \mathbb{N} \mid n \text{ satisfies } p\}$$

In a linear structure, the natural numbers $0,1,2\dots$ represent the time instants, and the states in them, and S represents, for every proposition, the states where it holds, or, alternatively, it represents for every state the propositions it satisfies. The temporal operators of LTL allows to quantify (existentially and universally) w.r.t. the traversed states. To define the satisfaction relation, we need to check properties on future states, like some sort of "time travel." To this aim we define the following *shifting* operation on S.

Definition 12.3 (Shifting). Let (S,P) be a linear structure. For any natural number k we let (S^k,P) denote the linear structure where:

$$\forall p \in P.\ S^k(p) = \{n-k \mid n \geq k \ \wedge\ n \in S(p)\}$$

As done for the HM-logic, we define the a notion of satisfaction $\models$ as follows.

Definition 12.4 (LTL satisfaction relation). Given a linear structure (S,P) we define the satisfaction relation $\models$ for LTL formulas by structural induction:

$$
\begin{array}{ll}
S \models \mathit{true} & \\
S \models \neg\phi & \text{if it is not true that } S \models \phi \\
S \models \phi_0 \wedge \phi_1 & \text{if } S \models \phi_0 \text{ and } S \models \phi_1 \\
S \models \phi_0 \vee \phi_1 & \text{if } S \models \phi_0 \text{ or } S \models \phi_1 \\
S \models p & \text{if } 0 \in S(p) \\
S \models O\ \phi & \text{if } S^1 \models \phi \\
S \models F\ \phi & \text{if } \exists k \in \mathbb{N} \text{ such that } S^k \models \phi \\
S \models G\ \phi & \text{if } \forall k \in \mathbb{N} \text{ it holds } S^k \models \phi \\
S \models \phi_0\ U\ \phi_1 & \text{if } \exists k \in \mathbb{N} \text{ such that } S^k \models \phi_1 \text{ and } \forall i < k.\ S^i \models \phi_0
\end{array}
$$

Two LTL formulas ϕ and ψ are called *equivalent*, written $\phi \equiv \psi$ if for any S we have $S \models \phi$ iff $S \models \psi$. From the satisfaction relation it is easy to check that the operators F and G can be expressed in terms of the until operator as follows:

$$
\begin{aligned}
F\ \phi &\equiv \mathit{true}\ U\ \phi \\
G\ \phi &\equiv \neg(F\ \neg\phi) \ \equiv\ \neg(\mathit{true}\ U\ \neg\phi)
\end{aligned}
$$

In the following we let

$$\phi_0 \Rightarrow \phi_1 \stackrel{\text{def}}{=} \phi_1 \vee \neg\phi_0$$

denote the logical implication.

Other commonly used operators are *weak until* (W), *release* (R) and *before* (B). They can be derived as follows:

W: The formula $\phi_0\ W\ \phi_1$ is analogous to the ordinary "until" operator except for the fact that $\phi_0\ W\ \phi_1$ is also true when ϕ_0 holds always, i.e., $\phi_0\ U\ \phi_1$ requires that ϕ_1 holds sometimes in the future, while this is not necessarily the case for $\phi_0\ W\ \phi_1$. Formally, we have:

$$\phi_0\ W\ \phi_1 \stackrel{\text{def}}{=} (\phi_0\ U\ \phi_1) \ \vee\ G\ \phi_0$$

R: The formula $\phi_0\ R\ \phi_1$ asserts that ϕ_1 must be true until and including the point where ϕ_0 becomes true. As in the case of weak until, if ϕ_0 never becomes true, then ϕ_1 must hold always. Formally, we have:

$$\phi_0\ R\ \phi_1 \stackrel{\text{def}}{=} \phi_1\ W\ (\phi_1\ \wedge\ \phi_0)$$

B: The formula $\phi_0\ B\ \phi_1$ asserts that ϕ_0 holds sometime before ϕ_1 holds or ϕ_1 never holds. Formally, we have:

$$\phi_0\ B\ \phi_1 \stackrel{\text{def}}{=} \phi_0\ R\ \neg\phi_1$$

We can graphically represent a linear structure S as a diagram like

$$0 \to 1 \to \cdots \to k \to \cdots$$

where additionally each node can be tagged with some of the formulas it satisfies: we write $k_{\phi_1,\ldots,\phi_n}$ if $S^k \models \phi_1 \wedge \cdots \wedge \phi_n$.

For example, given $p,q \in P$, we can visualise the linear structures that satisfy some basic LTL formulas as follows:

$$\begin{array}{rl}
X\ p & 0 \to 1_p \to 2 \to \cdots \\
F\ p & 0 \to \cdots \to (k-1) \to k_p \to (k+1) \to \cdots \\
G\ p & 0_p \to 1_p \to \cdots \to k_p \to \cdots \\
p\ U\ q & 0_p \to 1_p \to \cdots \to (k-1)_p \to k_q \to (k+1) \to \cdots \\
p\ W\ q & \begin{cases} 0_p \to 1_p \to \cdots \to (k-1)_p \to k_q \to (k+1) \to \cdots \\ 0_p \to 1_p \to \cdots \to k_p \to \cdots \end{cases} \\
p\ R\ q & \begin{cases} 0_q \to 1_q \to \cdots \to (k-1)_q \to k_{p,q} \to (k+1) \to \cdots \\ 0_q \to 1_q \to \cdots \to k_q \to \cdots \end{cases} \\
p\ B\ q & \begin{cases} 0_{\neg q} \to 1_{\neg q} \to \cdots \to (k-1)_{\neg q} \to k_{\neg q,p} \to (k+1) \to \cdots \\ 0_{\neg q} \to 1_{\neg q} \to \cdots \to k_{\neg q} \to \cdots \end{cases}
\end{array}$$

We now show some examples that illustrate the expressiveness of LTL.

Example 12.1. Consider the following LTL formulas:

$G\ \neg p$:	p will never happen, so it is a safety property.
$p \Rightarrow F\ q$:	if p happens now then also q will happen sometime in the future.
$G\ F\ p$:	p happens infinitely many times in the future, so it is a fairness property.
$F\ G\ p$:	p will hold from some time in the future onward.

Finally, $G(req \Rightarrow (req\ U\ grant))$ expresses the fact that whenever a request is made it holds continuously until it is eventually granted.

12.2.2 Computation Tree Logic

In this section we introduce CTL and CTL*, two logics which use trees as models of time: computation is no longer deterministic along time, but at each instant some possible futures can be taken. CTL and CTL* extend LTL with two operators which allow to express properties on paths over trees. The difference between CTL and CTL* is that the former is a restricted version of the latter. So we start by introducing the more expressive logic CTL*.

12.2.2.1 CTL*

CTL* still includes the temporal operators O, F, G and U: they are called *linear operators*. However, it introduces two new operators, called *path operators*:

E: The formula $E\ \phi$ (to be read "possibly ϕ") means that there *exists* some path that satisfies ϕ. In the literature it is sometimes written $\exists\ \phi$.

A: The formula $A\ \phi$ (to be read "inevitably ϕ") means that each path of the tree satisfies ϕ, i.e., that ϕ is satisfied along *all* paths. In the literature it is sometimes written $\forall\ \phi$.

Definition 12.5 (CTL* formulas). The syntax of CTL* formulas is as follows:

$$\begin{aligned}\phi \ ::= \ & true \mid false \mid \neg\phi \mid \phi_0 \wedge \phi_1 \mid \phi_0 \vee \phi_1 \mid \\ & p \mid O\ \phi \mid F\ \phi \mid G\ \phi \mid \phi_0\ U\ \phi_1 \mid \\ & E\ \phi \mid A\ \phi\end{aligned}$$

where $p \in P$ is any atomic proposition.

In the case of CTL*, instead of using linear structures, the computation of the system over time is represented by using infinite trees as explained below.

We recall that a (possibly infinite) tree $T = (V, \to)$ is a directed graph with vertices in V and directed arcs given by $\to \subseteq V \times V$, where there is one distinguished vertex $v_0 \in V$ (called *root*) such that there is exactly one directed path from v_0 to any other vertex $v \in V$.

Definition 12.6 (Infinite tree). Let $T = (V, \to)$ be a tree, with V the set of nodes, v_0 the root and $\to \subseteq V \times V$ the parent-child relation. We say that T is an *infinite tree* if $\to$ is *total* on V, namely if every node has a child:

$$\forall v \in V.\ \exists w \in V.\ v \to w$$

Definition 12.7 (Branching structure). A *branching structure* is a triple (T, S, P), where P is a set of *atomic propositions*, $T = (V, \to)$ is an infinite tree and $S : P \to \wp(V)$ is a function from the atomic propositions to subsets of nodes of V defined as follows:

$$\forall p \in P.\ S(p) = \{x \in V \mid x \text{ satisfies } p\}$$

In CTL* computations are described as infinite paths on infinite trees.

Definition 12.8 (Infinite paths). Let $T = (V, \to)$ be an infinite tree and $\pi = v_0, v_1, \ldots$ be an infinite sequence of nodes in V. We say that π is an *infinite path* over T if

$$\forall i \in \mathbb{N}.\ v_i \to v_{i+1}$$

Of course, we can view an infinite path $\pi = v_0, v_1, \ldots$ as a function $\pi : \mathbb{N} \to V$ such that $\pi(i) = v_i$ for any $i \in \mathbb{N}$. As for the linear case, we need a shifting operators on paths.

Definition 12.9 (Path shifting). Let $\pi = v_0, v_1, \ldots$ be an infinite path over T and $k \in \mathbb{N}$. We let the infinite path π^k be defined as follows:

$$\pi^k = v_k, v_{k+1}, \ldots$$

In other words, for an infinite path $\pi : \mathbb{N} \to V$ we let $\pi^k : \mathbb{N} \to V$ be the function defined as $\pi^k(i) = \pi(k+i)$ for all $i \in \mathbb{N}$.

Definition 12.10 (CTL* satisfaction relation). Let (T,S,P) be a branching structure and $\pi = v_0, v_1, v_2, \ldots$ be an infinite path. We define the satisfaction relation $\models$ inductively as follows:

- state operators:

$S,\pi \models \mathit{true}$	
$S,\pi \models \neg\phi$	if it is not true that $S,\pi \models \phi$
$S,\pi \models \phi_0 \wedge \phi_1$	if $S,\pi \models \phi_0$ and $S,\pi \models \phi_1$
$S,\pi \models \phi_0 \vee \phi_1$	if $S,\pi \models \phi_0$ or $S,\pi \models \phi_1$
$S,\pi \models p$	if $v_0 \in S(p)$
$S,\pi \models O\ \phi$	if $S,\pi^1 \models \phi$
$S,\pi \models F\ \phi$	if $\exists i \in \mathbb{N}$ such that $S,\pi^i \models \phi$
$S,\pi \models G\ \phi$	if $\forall i \in \mathbb{N}$ it holds $S,\pi^i \models \phi$
$S,\pi \models \phi_0\ U\ \phi_1$	if $\exists i \in \mathbb{N}$ such that $S,\pi^i \models \phi_1$ and $\forall j < i.\ S,\pi^j \models \phi_0$

- path operators:[2]

$S,\pi \models E\phi$	if there exists $\pi' = v_0, v_1', v_2', \ldots$ such that $S,\pi' \models \phi$
$S,\pi \models A\phi$	if for all paths $\pi' = v_0, v_1', v_2', \ldots$ we have $S,\pi' \models \phi$

Two CTL* formulas ϕ and ψ are called *equivalent*, written $\phi \equiv \psi$ if for any S,π we have $S,\pi \models \phi$ iff $S,\pi \models \psi$.

Example 12.2. Consider the following CTL* formulas:

$E\ O\ \phi$:	is analogous to the HM-logic formula $\Diamond\phi$.
$A\ G\ p$:	means that p happens in all reachable states.
$E\ F\ p$:	means that p happens in some reachable state.
$A\ F\ p$:	means that on every path there exists a state where p holds.
$E\ (p\ U\ q)$:	means that there exists a path where p holds until q.
$A\ G\ E\ F\ p$:	in every future exists a successive future where p holds.

12.2.2.2 CTL

The formulas of CTL are obtained by restricting CTL*. Let $\{O,F,G,U\}$ be the set of *linear operators*, and $\{E,A\}$ be the set of *path operators*.

[2] Note that in the case of path operators, only the first node v_0 of π is relevant.

Definition 12.11 (CTL formulas). A CTL* formula is a *CTL formula* if all of the followings hold:

1. each path operator appear only immediately before a linear operator;
2. each linear operator appears immediately after a path operator.

In other words, CTL allows only the combined use of path operators with linear operators, like in EO, AO, EF, AF, etc. It is evident that CTL and LTL are both[3] subsets of CTL*, but they are not equivalent to each other. Without going into the detail, we mention that:

- no CTL formula is equivalent to the LTL formula $F\ G\ p$;
- no LTL formula is equivalent to the CTL formula $AG\ (p \Rightarrow (EO\ q \wedge EO\ \neg q))$.

Moreover, fairness is not expressible in CTL.

Finally, we note that all CTL formulas can be written in terms of the minimal set of operators *true*, $\neg$, $\vee$, EG, EU, EO. In fact, for the remaining (combined) operators we have the following logical equivalences:

$$\begin{aligned}
EF\phi &\equiv E(true\ U\ \phi)\\
AO\phi &\equiv \neg(EO\neg\phi)\\
AG\phi &\equiv \neg(EF\neg\phi)\ \equiv\ \neg E(true\ U\ \neg\phi)\\
AF\phi &\equiv A(true\ U\ \phi)\ \equiv\ \neg(EG\neg\phi)\\
A(\phi\ U\ \varphi) &\equiv \neg(E(\neg\varphi\ U\ \neg(\phi\vee\varphi))\vee EG\neg\varphi)
\end{aligned}$$

Example 12.3. All the CTL* formulas in Example 12.2 are also CTL formulas.

12.3 μ-Calculus

Now we introduce the μ-calculus. The idea is to add the least and greatest fixpoint operators to modal logic. We remark that HM-logic was introduced not so much as a language to write down system specifications, but rather as an aid to understanding process equivalence from a logical point of view. As a matter of fact, many interesting properties of reactive systems can be conveniently expressed as fixpoints. The two operators that we introduce are the following:

$\mu x.\ \phi$: is the least fixpoint of the equation $x \equiv \phi$.
$\nu x.\ \phi$: is the greatest fixpoint of $x \equiv \phi$.

As a rule of thumb, we can think that least fixpoints are associated with liveness properties, while greatest fixpoints with safety properties.

[3] An LTL formula ϕ is read as the CTL* formula $A\phi$. Namely, the structure where a LTL formula is evaluated corresponds to a CTL* tree consisting of a set of traces.

Definition 12.12 (μ-calculus formulas). The syntax of μ-calculus formulas is:

$$\begin{aligned}\phi \ ::= \ & \mathit{true} \mid \mathit{false} \mid \phi_0 \wedge \phi_1 \mid \phi_0 \vee \phi_1 \mid \\ & p \mid \neg p \mid x \mid \Diamond \phi \mid \Box \phi \mid \mu x.\ \phi \mid \nu x.\ \mu \phi\end{aligned}$$

where $p \in P$ is any atomic proposition and $x \in X$ is any predicate variable.

In the following, we let $\mathscr{F}$ denote the set of μ-calculus formulas. To limit the number of parentheses and ease readability of formulas, we tacitly assume that modal operators have higher precedence than logical connectives, and that fixpoint operators have lowest precedence, meaning that the scope of a fixpoint variable extends as far to the right as possible.

The idea is to interpret formulas over a transition system (with vacuous transition labels): to each formula we associate the set of states of the transition system where the formula holds true. Then, the least and greatest fixpoint corresponds quite nicely to the notion of smallest and largest set of states where the formulas holds, respectively.

Since the powerset of the set of states is a complete lattice, in order to apply the fixpoint theory we require that the semantics of any formula ϕ is defined using monotone transformation functions This is the reason why we do not include general negation in the syntax, but only in the form $\neg p$ for p an atomic proposition. This way, provided that all recursively defined variables are distinct, the μ-calculus formulas we use are said to be in *positive normal form*. Alternatively, we can allow general negation and then require that in well-formed formulas any occurrence of a variable x is preceded by an even number of negations. Then, any such formula can be put in positive normal form by using De Morgan's laws, double negation ($\neg\neg\phi \equiv \phi$) and dualities:

$$\neg\Diamond\phi \equiv \Box\neg\phi \qquad \neg\Box\phi \equiv \Diamond\neg\phi \qquad \neg\mu x.\ \phi \equiv \nu x.\ \neg\phi[^{\neg x}/_{x}] \qquad \neg\nu x.\ \phi \equiv \mu x.\ \neg\phi[^{\neg x}/_{x}]$$

Let $(V, \to)$ be an LTS (with vacuous transition labels), X be the set of predicate variables and P be a set of propositions, we introduce a function $\rho : P \cup X \to \wp(V)$ which associates to each proposition and to each variable a subset of states of the LTS. Then we define the denotational semantics of μ-calculus which maps each μ-calculus formula ϕ to the subset of states $[\![\phi]\!]\rho$ in which it holds (according to ρ).

Definition 12.13 (Denotational semantics of the μ-calculus). We define the interpretation function $[\![\cdot]\!] : \mathscr{F} \to (P \cup X \to \wp(V)) \to \wp(V)$ by structural recursion on formulas as follows:

$$
\begin{aligned}
[\![true]\!]\,\rho &= V\\
[\![false]\!]\,\rho &= \varnothing\\
[\![\phi_0 \wedge \phi_1]\!]\,\rho &= [\![\phi_0]\!]\,\rho \cap [\![\phi_1]\!]\,\rho\\
[\![\phi_0 \vee \phi_1]\!]\,\rho &= [\![\phi_0]\!]\,\rho \cup [\![\phi_1]\!]\,\rho\\
[\![p]\!]\,\rho &= \rho(p)\\
[\![\neg p]\!]\,\rho &= V \setminus \rho(p)\\
[\![x]\!]\,\rho &= \rho x\\
[\![\Diamond\phi]\!]\,\rho &= \{\, v \mid \exists v' \in [\![\phi]\!]\,\rho.\ v \to v' \,\}\\
[\![\Box\phi]\!]\,\rho &= \{\, v \mid \forall v'.\ v \to v' \Rightarrow v' \in [\![\phi]\!]\,\rho \,\}\\
[\![\mu x.\ \phi]\!]\,\rho &= \mathrm{fix}\ \lambda S.\ [\![\phi]\!]\,\rho[^S/_x]\\
[\![\nu x.\ \phi]\!]\,\rho &= \mathrm{FIX}\ \lambda S.\ [\![\phi]\!]\,\rho[^S/_x]
\end{aligned}
$$

where FIX denotes the greatest fixpoint.

The definitions are straightforward. The only equations that need some comments are those related to the modal operators $\Diamond\phi$ and $\Box\phi$: in the first case, we take as $[\![\Diamond\phi]\!]\,\rho$ the set of states v that have (at least) one transition to a state v' that satisfies ϕ; in the second case, we take as $[\![\Box\phi]\!]\,\rho$ the set of states v such that all outgoing transitions lead to some states v' that satisfy ϕ. Note that, as a particular case, a state with no outgoing transitions trivially satisfy the formula $\Box\phi$ for any ϕ. For example the formula $\Box$*false* is satisfied by all and only deadlock states; vice versa $\Diamond$*true* is satisfied by all and only non-deadlock states. Intuitively, we can note that the modality $\Diamond\phi$ is somewhat analogous to the CTL formula $EO\ \phi$, while the modality $\Box$ can play the role of $AO\ \phi$.

Fixpoints are computed in the $\mathrm{CPO}_\perp$ of sets of states, ordered by inclusion: $(\wp(V), \subseteq)$. Union and intersections are of course monotone functions. Also the functions associated with modal operators

$$
\lambda S.\ \{\, v \mid \exists v' \in S.\ v \to v' \,\} \qquad \lambda S.\ \{\, v \mid \forall v'.\ v \to v' \Rightarrow v' \in S \,\}
$$

are monotone. The least fixpoint of a function $f : \wp(V) \to \wp(V)$ can then be computed by taking the limit $\bigcup_{n\in\mathbb{N}} f^n(\varnothing)$, while for the greatest fixpoint, we take $\bigcap_{n\in\mathbb{N}} f^n(V)$. In fact, when f is monotone, we have:

$$
\varnothing \subseteq f(\varnothing) \subseteq f^2(\varnothing) \subseteq \cdots \subseteq f^n(\varnothing) \subseteq \cdots
$$

$$
V \supseteq f(V) \supseteq f^2(V) \supseteq \cdots \supseteq f^n(V) \supseteq \cdots
$$

Example 12.4 (Basic examples). Let us consider the following formulas:

$\mu x.\ x$: $\quad [\![\mu x.\ x]\!]\,\rho \stackrel{\mathrm{def}}{=} \mathrm{fix}\ \lambda S.\ S = \varnothing$.

In fact, let us approximate the result in the usual way:

$$
S_0 = \varnothing \qquad S_1 = (\lambda S.\ S)S_0 = S_0
$$

$\nu x.\, x$: $[\![\nu x.\, x]\!]\rho \stackrel{\text{def}}{=} \text{FIX } \lambda S.\, S = V$.
In fact, we have

$$S_0 = V \qquad S_1 = (\lambda S.\, S)S_0 = S_0$$

$\mu x.\, \Diamond x$: $[\![\mu x.\, \Diamond x]\!]\rho \stackrel{\text{def}}{=} \text{fix } \lambda S.\, \{v \mid \exists v' \in S.\, v \to v'\} = \varnothing$.
In fact, we have:

$$S_0 = \varnothing \qquad S_1 = \{v \mid \exists v' \in \varnothing.\, v \to v'\} = \varnothing$$

$\mu x.\, \Box x$: $[\![\mu x.\, \Box x]\!]\rho \stackrel{\text{def}}{=} \text{fix } \lambda S.\, \{v \mid \forall v'.\, v \to v' \Rightarrow v' \in S\}$.
By successive approximations, we get:

$$\begin{aligned}
S_0 &= \varnothing \\
S_1 &= \{v \mid \forall v'.\, v \to v' \Rightarrow v' \in \varnothing\} = \{v \mid v \not\to\} \\
&= \{v \mid v \text{ has no outgoing arc}\} \\
S_2 &= \{v \mid \forall v'.\, v \to v' \Rightarrow v' \in S_1\} \\
&= \{v \mid v \text{ has outgoing paths of length at most } 1\} \\
&\dots \\
S_n &= \{v \mid \forall v'.\, v \to v' \Rightarrow v' \in S_{n-1}\} \\
&= \{v \mid v \text{ has outgoing paths of length at most } n-1\}
\end{aligned}$$

We can conclude that $[\![\mu x.\, \Box x]\!]\rho = \bigcup_{i\in\mathbb{N}} S_i$ is the set of vertices whose outgoing paths have all finite length.

$\nu x.\, \Box x$: $[\![\nu x.\, \Box x]\!]\rho \stackrel{\text{def}}{=} \text{FIX } \lambda S.\, \{v \mid \forall v', v \to v' \Rightarrow v' \in S\} = V$.
In fact, we have:

$$S_0 = V \qquad S_1 = \{v \mid \forall v'.\, v \to v' \Rightarrow v' \in V\} = V$$

$\mu x.\, p \vee \Diamond x$: $[\![\mu x.\, p \vee \Diamond x]\!]\rho \stackrel{\text{def}}{=} \text{fix } \lambda S.\, \rho(p) \cup \{v \mid \exists v' \in S.\, v \to v'\}$.
Let us compute some approximations:

$$\begin{aligned}
S_0 &= \varnothing \\
S_1 &= \rho(p) \\
S_2 &= \rho(p) \cup \{v \mid \exists v' \in \rho(p).\, v \to v'\} \\
&= \{v \mid v \text{ can reach some } v' \in \rho(p) \text{ in less than one step}\} \\
&\dots \\
S_n &= \{v \mid v \text{ can reach some } v' \in \rho(p) \text{ in less than } n-1 \text{ steps}\} \\
&\dots \\
\bigcup_{n\in\mathbb{N}} S_n &= \{v \mid v \text{ has a finite path to some } v' \in \rho(p)\}
\end{aligned}$$

Thus, the formula is similar to the CTL formula $EF\ p$, meaning that some node in $\rho(p)$ is reachable.

The μ-calculus is more expressive than CTL* (and consequently than CTL and LTL), in fact all CTL* formulas can be translated to μ-calculus formulas. This makes the μ-calculus probably the most studied of all temporal logics of programs. Unfortunately, the increase in expressive power we get from μ-calculus is balanced in an equally great increase in awkwardness: we invite the reader to check by her/himself how relatively easy is to write down short μ-calculus formulas whose intended meanings remain obscure after several attempts to decipher them. Still, many correctness properties can be expressed in a very concise and elegant way in the μ-calculus. The full translation from CTL* to μ-calculus is quite complex and we do not account for it here.

Example 12.5 (More expressive examples). Let us now briefly discuss some more complicated examples:

$\mu x.\ (p \wedge \Diamond x) \vee q$:	it corresponds to the CTL formula $E(p\ U\ q)$.
$\mu x.\ (p \wedge \Box x \wedge \Diamond x) \vee q$:	it corresponds to the LTL/CTL formula $A(p\ U\ q)$. Note that in this case the sub-formula $\Diamond x$ is necessary to discard deadlock states.
$\nu x.\ \mu y.\ (p \wedge \Diamond x) \vee \Diamond y$:	it corresponds to the CTL* formula $EGF\,p$: given a path, $\mu y.\ (p \wedge \Diamond x) \vee \Diamond y$ means that after a finite number of steps you find a vertex where both: (1) p holds, and (2) you can reach a vertex where the property recursively holds.

Without increasing the expressive power of μ-calculus, formulas can be extended to deal with labelled transitions, in the style of extending HM-logic with recursion (see Problem 12.10).

12.4 Model Checking

The problem of model checking consists in the exaustive, possibly automatic, verification of whether a given model of a system meets or not a given logic specification of the properties the system should satisfy, like absence of deadlocks.

The main ingredients of model checking are:

- an LTS M (the model) and a vertex v (the initial state);
- a formula ϕ (in temporal or modal logic) you want to check.

The problem of model checking is: *does v in M satisfy ϕ?*

The result of model checking should be either a positive answer or some counterexample explaining one possible reason why the formula is not satisfied.

Without entering in the details, one successful approach to model checking consists of: 1) computing a finite LTS $M_{\neg\phi}$ that is to some extent equivalent to the negation of the formula ϕ under inspection; roughly, each state in the constructed

LTS represents a set of LTL formulas that hold from that state; 2) computing some form of product between the model M and the computed LTS $M_{\neg\phi}$; roughly, this corresponds to solving a non-emptiness problem for the intersection of (the languages associated with) M and $M_{\neg\phi}$; 3) if the intersection is non-empty, then a finite witness can be constructed that offers a counterexample to the validity of the formula ϕ in M.

In the case of μ-calculus formulas, fixpoint theory gives a straightforward (iterative) implementation for a model checker by computing the set of all and only states that satisfy a formula by successive approximations. In model checking algorithms, it is often convenient to proceed by evaluating formulas with the aid of dynamic programming. The idea is to work in a bottom-up fashion: starting from the atomic predicates that appear in the formula, we mark all the states with the sub-formulas they satisfy. When a variable is encountered, a separate activation of the procedure is allocated for computing the fixpoint of the corresponding recursive definition.

For computing a single fixpoint, the length of the iteration is in general transfinite but is bounded at worst by the cardinal after cardinality of the lattice and in the special case of $\wp(V)$ by the cardinal after the cardinality of V. In practice, many systems can be modelled, at some level of abstraction, as finite state systems, in which case a finite number of iterations ($|V|+1$ at worst) suffices. When two or more fixpoints of the same kind are nested within each other, then we can exploit monotonicity to avoid restarting the computation of the innermost fixpoint at each iteration of the outermost one. However, when least and greatest fixpoints are nested in alternation, this optimisation is no longer possible and the time needed to model check the formula is exponential w.r.t. the so called *alternation depth* of fixpoints in the formula.

From a purely theoretical perspective, the hierarchy obtained by considering formulas ordered according to the alternation depth of fixpoint operators gives more expressive power as the alternation depth increases: model checking in the μ-calculus is proved to be in $\mathsf{NP} \cap \mathsf{coNP}$ (μ-calculus is closed under complementation).

From a pragmatic perspective, any reasonable specification requires at most alternation depth 2 (i.e., it is unlikely to find correctness properties that require alternation depth equal or higher than 3). Moreover, the dominant factor in the complexity of model checking is typically the size of the model rather than the size of the formula, because specifications are often very short: sometimes even exponential growth in the specification size can be tolerable. For these reasons, in many cases, the before mentioned, complex translation from CTL* formulas to μ-calculus formulas is able to guarantee competitive model checking.

In the case of reactive systems, the LTS is often given implicitly, as the one associated with a term of some process algebra, because in this way the structure of the system is handled more conveniently. However, as noted in the previous chapter, even for finite processes, the size of their actual LTS can explode.

When it becomes unfeasible to represent the whole set of states, one approach is to use *abstraction* techniques. Roughly, the idea is to devise a smaller, less detailed model by suppressing inessential data from the original, fully detailed model. Then, as far as the correctness of the larger model follows from the correctness of the smaller model, we are guaranteed that the abstraction is sound.

One possibility to tackle the state explosion problem is to minimise the system according to some suitable equivalence. Note that minimisation can take place also while combining subprocesses and not just at the end. Of course, this technique is viable only if the minimisation preserves all properties to be checked. For example, the validity of any μ-calculus formula is invariant w.r.t. bisimulation, thus we can minimise LTSs up to bisimilarity before model checking them.

Another important technique to succinctly represent large systems is to take a *symbolic* approach, like representing the sets of states where formulas are true in terms of their boolean characteristic functions, expressed as ordered *Binary Decision Diagrams* (BDDs). This approach has been very successful for the debugging and verification of hardware circuits, but, for reasons not well understood, software verification has proved more elusive, probably because programs lack some form of regularity that commonly arises in electronic circuits. In the worst case, also symbolic techniques can lead to intractably inefficient model checking.

Problems

12.1. Suppose there are two processes p_1 and p_2 that can access a single shared resource r. We are given the following atomic propositions, for $i = 1,2$:

req_i: holds when process p_i is requesting access to r;
use_i: holds when process p_i has had access to r;
rel_i: holds when process p_i has released r.

Use LTL formulas to specify the following properties:

1. mutual exclusion: r is accessed by only one process at a time;
2. release: every time r is accessed by p_i, it is released after a finite amount of time;
3. priority: whenever both p_1 and p_2 require access to r, p_1 is granted access first;
4. absence of starvation: whenever p_i requires access to r, it is eventually granted access to it.

12.2. Consider an elevator system serving three floors, numbered 0 to 2. At each floor there is an elevator door that can be open or closed, a call button, and a light that is on when the elevator has been called. Define a set of atomic propositions, as small as possible, to express the following properties as LTL formulas:

1. a door in not open if the elevator is not present at that floor;
2. every elevator call will be served;
3. every time the elevator serves a floor the corresponding light is turned off;
4. the elevator will always return to floor 0;
5. a request at the top floor has priority over all the other requests.

12.3. Consider the CTL* formula $\phi \stackrel{\text{def}}{=} AF\ G\ (p \vee O\ q)$. Explain the property associated with it and define a branching structure where it is satisfied. Is it a LTL formula? Is it a CTL formula?

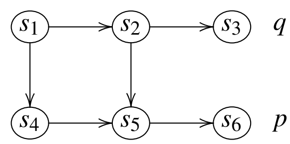
12.4. Prove that if the CTL* formula $AO\ \phi$ is satisfied, then also the formula $O\ A\ \phi$ is satisfied. Is the converse true?

12.5. Is it true that the CTL* formulas $A\ G\ \phi$ and $G\ A\ \phi$ are logically equivalent?

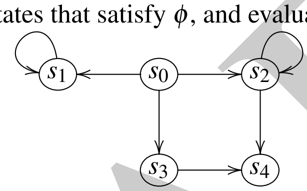
12.6. Given the μ-calculus formula:

$$\phi \stackrel{\text{def}}{=} \nu x.\ (p \vee \Diamond x) \wedge (q \vee \Box x)$$

compute its denotational semantics and evaluate it on the LTS below:



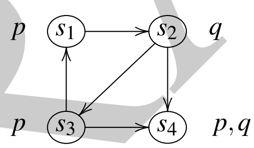
12.7. Given the μ-calculus formula $\phi \stackrel{\text{def}}{=} \nu x.\ \Diamond x$, compute its denotational semantics, spelling out what are the states that satisfy ϕ, and evaluate it on the LTS below:



12.8. Write a μ-calculus formula ϕ representing the statement:

'p is always true along any path leaving the current state.'

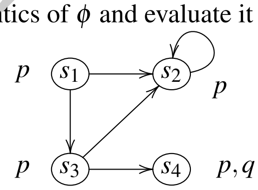
Write the denotational semantics of ϕ and evaluate it over the LTS below:



12.9. Write a μ-calculus formula ϕ representing the statement:

'there is some path where p holds until eventually q holds.'

Write the denotational semantics of ϕ and evaluate it over the LTS below:



12.10. Let us extend the μ-calculus with the formulas $\langle A \rangle \phi$ and $[A]\phi$, where A is a set of labels: they represent, respectively, the ability to perform a transition with some label $a \in A$ and reach a state that satisfies ϕ, and the necessity to reach a state that satisfies ϕ after performing any transition with label $a \in A$.

1. Define the semantics $[\![\langle A\rangle\phi]\!]\,\rho$ and $[\![[A]\phi]\!]\,\rho$.
2. Let us write $\langle a_1,\dots,a_n\rangle\phi$ and $[a_1,\dots,a_n]\phi$ in place of $\langle\{a_1,\dots,a_n\}\rangle\phi$ and $[\{a_1,\dots,a_n\}]\phi$, respectively. Compute the denotational semantics of the formulas

$$\phi_1 \stackrel{\text{def}}{=} \nu x.\ ((\langle a\rangle true \wedge \langle b\rangle true) \vee p) \wedge [a,b]x \qquad \phi_2 \stackrel{\text{def}}{=} \mu x.\ p \vee \langle a,b\rangle x$$

   and evaluate them on the LTS below: