Modelling of Complex Systems

Marc Denecker

January 3, 2022

Contents

Chapter 1

Introduction

Judging by its title, this course is about modelling systems.

The word "system" should be understood in a broad sense: a system can be any domain involving a collection of any sorts of objects and sorts of relations between them. Time may be one of the type of objects, in which case evolving objects and relations may be part of the system. It then is a dynamic domain, including e.g., agents performing actions that change the world, such as a planning domain, or a piece of software. But a system may also be a static system, such as the design of a seal, the positioning of boxes in a container, a schedule, a study program, etc.

Such systems will be modelled. To "model" is a verb, used in this course as a synonym of terms such as "to specify", "to represent knowledge".

The result of this act is "a modelling", "a knowledge base", "a specification", "a logic theory". These are terms used in different research communities, but they will be synonyms here.

Perhaps the most evident place where modellings are built, is in empirical formal sciences. These are sciences that use mathematical theories to model an observable reality. E.g., we could say that Newton's theory of gravitation is a mathematical modelling of gravitation. A more standard term for "a modelling" is the word "a model", which is a term often used in engineering and empirical science, as in a "a climate model". However, we cannot use this term here in this course, since in the context of logic, the word "a model" has a different meaning from how it is used elsewhere in science, and we will use the term "a model" in the logical sense.

The modellings in this course are expressed in logic. The course will study several of logics in some detail (Classical first order logic (FO), Linear Time Logic (LTL), Computation Tree Logic (CTL), event-B). As such, logics are an important topic of this course, and the course aims to give an introduction to several theoretical aspects of them: syntax, formal semantics, informal semantics, expressivity, different sorts of inference. If these terms are not clear to you now, I certainly hope they will be clearer at the end of the course.

Modellings, e.g., scientific theories, are clearly not programs: they are not executable, they cannot be run, they don't do anything. So, why then is a course on modelling a compulsory course in the Master of Computer Science? This is because modellings describe the application domain, and the information they contain may be useful to solve problems. One of the goals of this course is to develop an idea of how modellings can be used to solve a range of computational problems. That will bring us close to the topic of declarative programming languages (e.g., Prolog, Haskell) and even closer to declarative problem solving paradigms (e.g., constraint

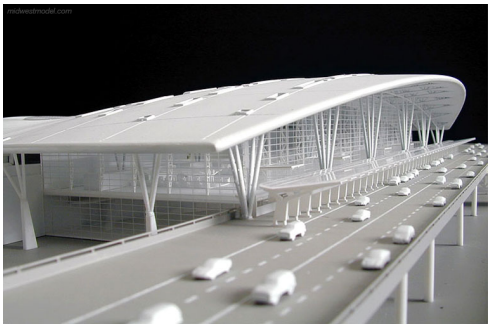


Figure 1.1: A scale model of the Indianapolis international airport

solving languages, semantic web languages).

The goals of the course are not only on the theoretical side. One goal is of a practical nature: to develop practical skills to use logics for modelling, and to develop an understanding how these modellings can be used to solve problems. These skills are mainly taught in the exercise sessions and in the project, so make sure that you attend these.

1.1 What is modelling?

A *modelling* is an artifact, made to capture certain aspects of some *application domain*, up to some level of abstraction. A modelling never reflects all aspects of the domain; it is an *abstraction* of the application domain. E.g., a scale model of a building does not model its internal structure, or the electrical wiring inside.

Examples of non-mathematical modellings that first come to mind are: scale models of a building (Figure 1.1), a blue print of a machine; a flow chart for a program; a road map. An example of a mathematical modelling is: a set of differential equations.

Sometimes, a modelling serves to describe an existing application domain. E.g., in empirical formal science, scientific theories are modellings of the existing empirical reality. Sometimes the application domain exists only in the imagination. E.g., a scale model of a building, or a blueprint of a machine are typically developed before the objects that they represent become real (and they may never become real).

Modelling in formal science A formal empirical science is a science that studies existing, observable ("empirical") phenomena, and makes mathematical ("formal") modellings for it, e.g., sets of equations. It is the area by excellence where modellings are used. The prototypical such science is physics.

Let us have a look in more detail at some physical theory. As an example, we select a famous case: Newton's gravitation theory, limited to two bodies. The application domain is that of two bodies b_1, b_2 that freely move in each others gravitation field. [1] Newtons law can be informally

[1] Newtons theory extends to an arbitrary number of bodies, but the two-body problems is *approximately* correct in a broad range of real situations, namely when the gravitation of one object dominates that of all other. E.g., the gravitation of the earth on the apple or on the moon dominates the gravitation of the sun and planets on them. The gravitation of the sun on the earth dominates that of all other planets.

represented as: *The acceleration of b_i is proportional to the mass of b_j ($i \neq j$) and inverse proportional to the square of the distance.* Stated in mathematical equations,

$$\frac{d^2P_1(t)}{d^2t}(t) = G \times \frac{m_2}{||P_2(t)-P_1(t)||^2}$$
$$\frac{d^2P_2(t)}{d^2t}(t) = G \times \frac{m_1}{||P_2(t)-P_1(t)||^2}$$

where bodies b_i are abstracted as pairs (m_i, P_i) with $m_i \in \mathbb{R}^+$ representing the mass of b_i and $P_i : \mathbb{R}^+ \to \mathbb{R}^3$ representing the trajectory of b_i, i.e., the position as a function of time.

Let us analyze this modelling: what are its ingredients, and how does it represent information about the real world. The same ingredients are found in all formal scientific theories and we will see that they occur also in modelling logics.[2]

We begin with the mathematical side of it.

- **Symbols**: the two equations are presented to us as *strings* of symbols. Different sorts of symbols can be distinguished:
  - **Mathematical or physical constants** have a fixed and known value. E.g., the symbols $\times, -, \div$, refer to the well-known numerical operators, $||\cdot||$ is the distance function defined on the cartesian space $\mathbb{R}^3$, $\frac{d^2}{d^2t}$ is the second derivative, etc.
    There is also the physical constant G called the *gravitational constant* ($\approx 6,67 \times 10^{-11} m^3 kg^{-1} s^{-2}$).
  - **Ontology symbols** are symbols that serve to represent the objects of the application domain. Here, these are the symbols m_i, P_i representing mass and position of the two bodies. Clearly, these symbols have a fixed interpretation in the domain, but their value is not given. They are often called the *variables* of the theory, but in this course, they will be called ontology symbols. The reason is that there is another type of symbol that is even more *variable*.
  - **Quantified variables** are symbols without fixed value; they are quantified over some set and their value ranges over that set. Here, t is a quantified variable implicitly quantified over $\mathbb{R}^+$.
- **Values**: symbols are strings but they refer to *values*, which are not strings.
  - E.g., the symbol G refers to the given real number, P_i refers to an unknown function $\mathbb{R} \to \mathbb{R}^3$, the symbol $\frac{d^2.}{d^2t}$ refers to the well-known second derivative operator.
- **Structures** are joint assignments of values to the ontology symbols. A structure is also called an *interpretation* (of the set of ontology symbols).
  - Here, a structure would be an assignment of values to the symbols m_1, P_1, m_2, P_2. Any assignment of positive real numbers to m_1, m_2 and of functions in $\mathbb{R}^+ \to \mathbb{R}^3$ to P_1, P_2 is a structure.
    The class of structures is immense and includes many thoroughly impossible ones such as structures representing zigzagging or teleporting trajectories, ...
- **Solutions** of the equations are structures that satisfy the theory.

[2]For the sake of the argument, the discussion below takes an idealized and simplified view on science. In particular, we ignore the fact that science needs to work with approximations. E.g., mass, time, position can be determined only approximately, and Newtons theory is approximately correct.

– A solution is a structure that satisfies the differential equations. I.e., such that for each $t \in \mathbb{R}^+$, the values left and right in each equation are identical.

There are infinitely many solutions to Newtons equations, but the set of solutions is incomparably much smaller than the set of structures. Given values representing an initial state of two bodies (i.e., values for m_1, m_2, $P_1(0), P_2(0)$, and for all derivatives $\frac{d^n P_1(t)}{d^n t}(0), \frac{d^n P_2(t)}{d^n t}(0)$), there are uncountably many structures expanding this initial state while there is only one unique solution expanding it.

A set of mathematical equations may be an interesting mathematical object, but it is meaningless as a scientific theory unless we know the **abstraction relation** between math and reality. We call this relation the **informal interpretation** and denote it as $\mathcal{I}$.

- Ontology symbols and values are **abstractions** of concepts, relations, attributes and objects in the application domain.

  E.g., in Newtons theory, physical bodies are abstracted as pairs (m, P) of a mass m and a trajectory function P. The informal interpretation of the ontology symbol m_2 is the concept of the mass of body b_2. The informal interpretation of the real number 54.4 (used as a mass) is a weight of 54.4kg. A structure $\mathfrak{A} = [m_1 : 78, P_1 : (\lambda t \in \mathbb{R}^+ : (0,0,0)), m_2 : 54.4, P_2 : (\lambda t \in \mathbb{R}^+ : (t,0,0))]$[3] is an abstraction of a state of affairs where b_1 has mass 78kg, sits still in the center of the universe, while b_2 has mass 54.4, is moving away from the center along the x-axis with a velocity of 1 meter per second. This structure does not satisfy the theory. [4].

  Science cannot exist without a precise understanding of these informal interpretations. Suppose an astronomer interprets b_1 as the sun and b_2 as Uranus and calculates real numbers t and x, y, z such that $P_2(t) = (x, y, z)$. To observe Uranus, he should be able to translate the number t to a specific time point (say, tonight at 20:34:51), and to translate the triple (x, y, z) to a specific direction for his telescope and he should look through his telescope, at that time, in that direction. This is far from trivial, and must be done with great precision. For this, the astronomer needs an extremely precise understanding of the relation between mathematical objects and the entities they stand for in the application domain.

  In general, the informal interpretation $\mathcal{I}$ assigns concepts of the application domain to ontology symbols σ, it assigns entities in the application domain to mathematical values v. From there on, it assigns interpretation to mathematical compound objects: structures as states of affairs of the domain, mathematical formulas as propositions about the domain.

  The informal interpretation exists only in our mind. The mathematical theory does not determine or constrain the informal interpretation $\mathcal{I}$. E.g., we normally interpret m_i as the mass of b_i, but nothing stops us from interpreting it as b_i's age, and the 3 coeficients of $P_i(t)$ as respectively the total mass, the total mass of gold and the total mass of silver in the body b_i. But what use would it have? Under this alternative informal interpretation, the equations do not hold in the real world and have zero predictive power. [5]

[3] Here $\lambda t \in \mathbb{R}^+ : (t,0,0)$ is a shorthand notation for the function $\mathbb{R}^+ \to \mathbb{R}^3 : t \mapsto (t,0,0)$.

[4] It is remarkable how much information is abstracted away in Newtons theory. E.g., the exact size and form and density of a spatial object, its geological and chemical structure; in case of an astronaut, the education, age, gender, ... None of these matters for the trajectory. For us, this seems evident but we have had 300 years to get use to. Recall that in the middle ages, planets were seen as divine objects, orchestrated by the 9 muses in a celestial harmony (google "Musica universalis" if you want to know more). Imagine the excitement when it was discovered that with so few variables, such precise predictions could be made.

[5] In my limited physical knowledge, I am aware of only one physical equation that has multiple, physically very

The importance of the informal interpretation in science and modelling cannot be stressed enough. Scientists depend on a common, precise understanding of the abstraction relation between symbols and their mathematical values on the one hand, and the real objects they study on the other hand. Without it, there is no science, there is only mathematics. Likewise, the informal interpretation/abstraction relation is of critical importance in all modelling applications. Without a well-understood informal interpretation, there is no modelling.

Therefore, any formal science consists not only of a set of mathematical equations or statements, but also of an *informal theory* describing the informal interpretation, i.e., the connection between the formal objects and the reality. E.g., in physics the informal theory will describe a.o. different ways to measure the mass of an object, time, position of objects. Because the reality is not a mathematical object, the informal theory cannot be entirely a mathematical theory. Not everything in formal science is mathematics.

- Given the informal interpretation $\mathcal{I}$, **structures** are abstractions of **potential states of affairs**.

  E.g., a potential state of affairs is that the sun (b_1) and the earth (b_2) are zigzagging through space; or that every second they switch positions. Structures exist that describe such states of affairs. But we know they are not possible.

- A scientific theory consisting of a mathematical theory T and informal interpretation $\mathcal{I}$ gives us **information** on the application domain. The mathematical theory determines which structures are solutions and which are non-solutions. The informal semantics allows us to translate this in terms of potential states of affairs: namely to decide which are possible and which are impossible. E.g., any structure with zigzagging and teleporting P_i's is a non-solution of Newtons theory. Hence, this theory tells us that planets do not zigzag or teleport under influence of gravitation.

  Solutions of T are abstractions of **possible states of affairs** (under $\mathcal{I}$), according to the scientific theory.

  Non-solutions of T are abstractions of **impossible states of affairs** (under $\mathcal{I}$) according to the scientific theory.

- In the other direction, a scientific theory is **true** in a state of affairs if the abstraction of that state of affairs (under $\mathcal{I}$) is a solution of the mathematical theory. Otherwise, it is **false**. Scientific experiments serve to discover if a scientific theory is true in the actual state of affairs.

- Typically, a scientific theory does not serve to describe a unique actual state of affairs, but a class $\mathcal{C}$ of possible states of affairs. A scientific theory T under $\mathcal{I}$ is **correct** if the theory is true in each state of affairs of $\mathcal{C}$ and false elsewhere. E.g., Newtons two-body gravitation theory aims to model all two body systems (sun-planet, planet-moon, ...). It is not correct. But under certain conditions, it is approximately correct. [6]

We see that a scientific theory distinguishes between solutions and non-solutions and, through the informal interpretation $\mathcal{I}$, between possible and impossible states of affairs. **This is the way a scientific theory conveys information about the application domain.**

different informal interpretations $\mathcal{I}$: the equations of harmonic oscillators. In different informal interpretations, the same symbol is interpreted as the mass of an object, the inertia, the inductance, the capacitance. These are all very different sorts of physical entities. See `https://en.wikipedia.org/wiki/Harmonic_oscillator`.

[6] These conditions include: that orbits are not significantly influenced by other bodies (e.g., Jupiter), or by other forces (e.g., electric or magnetic forces) or by relativistic effects bending space-time.

Following logical terminology, we will call a solution of a scientific theory a **model** of that theory. Hence, if the scientific theory is correct, a model of a theory is the abstraction of a possible state of affairs (according to that theory, under the given informal interpretation $\mathcal{I}$).

Derived semantical concepts Scientists and mathematicians use basic semantic concepts to describe theories, propositions and relations between them.

- **Consistency**: a scientific theory T is **consistent** if it has a solution (a model).

  Newtons theory is consistent. It has solutions; stronger, it has a solution for each given initial state. In the context of logic, a theory with models is called **satisfiable**.

- **Entailment**: a scientific theory T **entails** a proposition φ if φ is true in every model of T.

  E.g., Newtons theory entails that (under certain initial conditions) trajectories are elliptic. What this means is that in every model of the theory (where the initial conditions are met), the trajectories are elliptic functions.

- **Equivalence**: two scientific theories are **equivalent** if they have the same models.

These semantic concepts are fundamental concepts in formal science and mathematics. They are defined in terms of the distinction between models and non-models.

Also in logic, the concepts of consistency (often called satisfiability), entailment and (logical) equivalence are key concepts. It is important to understand that these concepts were not invented in logic, but logic merely copied these definitions from mathematics and formal science.

Calculemus! Models are *solutions* of the scientific theory. To computer scientists, the word "solutions" rings a bell. Could we say that a scientific theory is an encoding of a computational problem? That the models of the theory are the answer(s) to the problem? This is not a correct view in my opinion. To scientists, a scientific theory is a representation of knowledge, of information. Information is not a computational problem. In fact, one scientific theory induces many types of computational problems, the answers of which are all determined by the scientific theory and can be computed using a variety of mathematical techniques.

Here are some examples of problems that were solved using Newtons theory:

- *Computing a trajectory*, given a set of observed positions of the bodies.
- *Proving* that trajectories are elliptical.
- *Computing the escape velocity* of a rocket from earth.
- *Computing the distance for a geostationary orbit.*
- *Computing the position and mass of an unobserved planet* given some irregularities in the trajectory of another planet. [7]

[7] This is the way Neptunus was discovered. The physicist Le Verrier predicted its position from irregularities in the trajectory of Uranus, and Neptunus was discovered in 1846, exactly when and where Le Verrier had calculated it to be in 1845. It was a sensational moment of 19th-century science, and a dramatic confirmation of Newtonian gravitational theory: Le Verrier had discovered a planet "with the point of his pen.". See `https://en.wikipedia.org/wiki/Discovery_of_Neptune`.

As for the relevance for this course, the point is that a modelling is not a program, it is not even a specification of a computational problem, but it contains information, and this information can sometimes be used for solving a range or computational problems.

Information A scientific proposition or theory provides *information* about the empirical domain. But what is information? How can we get grip on such an ethereal topic?

Following ideas of the philosopher Wittgenstein, a piece of information is satisfied in some states of affairs and not in other states of affairs. The partition of potential state of affairs into the subclass that satisfies it, and the subclass that does not satisfy it, is to a high degree characteristic for the information: if two pieces of information yield the same partition, they convey the same information, otherwise they don't.

E.g., take two mathematical theories of gravitation. We can say that both contain the same information if they have the same solutions and the same non-solutions. Thus, the information in those theories is captured by the partition of structures in solutions and non-solutions.

This suggests a way to formalize information (or information content): as boolean functions on the class of all structures of ontology symbols. Sometimes, such a function is called an *infon*.

This suggests an informal technique, called *possible world analysis*, to analyze the meaning of propositions. Below two examples follow.

Example 1.1.1. Consider two statements: *(1) "All men love some film star"* versus *(2) "Some film star is loved by all men"*. Do they mean the same? I claim they do not and will show it by sketching a state of affairs in which (1) is true, and (2) is false. Take a state of affairs in which everybody except me loves Angelina Jolie and not Meryl Streep. I love only Meryl Streep. In this world the statement (1) is true: all men love some film star. On the other hand, (2) is not true. Hence, possible world analysis shows these statements are not equivalent.

Exercise 1.1.1. *We all know that the stress in a sentence can be important for the meaning of the sentence. Stress on a different word may result in a different meaning. The following natural language sentences only differ on where the stress is put. Say the sentences below and listen to yourself: you will hear that they mean something different. Apply a possible world analysis to show that they are not equivalent:*

- *"I" did not drive your car.*
- *I did not drive "your" car.*
- *I did not drive your "car".*

The stressed words are those between quotes. Note that in the three cases, the proposition is the same, but the stress adds an implicit message to the sentence. Such an implicit message is called an **implicature**. *E.g., the implicature of the first sentence is that "somebody else drove your car". Give the implicatures of the other two sentences. Then for each pair of these sentences, it suffices to find* one *state of affairs in which "I did not drive your car" but where the implicatures of both sentences disagree.*

Summary of concepts

Terminology 1.1.1.

- **domain**: a modelling is about an application domain. Within this domain, a number of distinguished types of objects, concepts and interrelations emerge as relevant to us. Our modellings focus on these relevant concepts and make abstraction of the rest. Below, the domain is sometimes called the *domain of discourse*: the domain about which we "speak".
- **ontology**: this is the set of relevant concepts of the domain.
- **state of affairs**: we can imagine the domain to be in one of many potential states of affairs. One state of affairs is characterized by the state of the concepts of the domain, i.e., the values of these concepts. Different states of affairs differ in the values that concepts have in it. Different states of affairs are mutually exclusive: if the domain is in one state of affairs, it is not in another.

  The domain may be real or imaginary. When the domain is real, the **actual state of affairs** is the one that is the case. Usually, we do not know the actual state of affairs, although we have information about it. Often the domain is imaginary. E.g., when we build a specification of software or hardware to be developed in the future.
- **information**: according to a piece of information, some states of affairs are **possible**, the others are **impossible**. If the state of affairs is possible we also say it **satisfies** that information. Vice versa, we say that a piece of information is **true** or **false** in a state of affairs. The information content of a proposition is characterized by the partition of possible and impossible states of affairs. Two pieces of information with the same possible and impossible states of affairs have the same information content. They are called **equivalent**.
- **proposition**: a statement (not necessarily in a logic) about the domain in terms of the given ontology, expresses a piece of information.
- **a modelling, a specification, a theory** : a set of statements in natural language or in mathematics or in some logic, providing propositions about the empirical domain aiming to describe the possible states of affairs of the domain. A modelling is **true** in a state of affairs if the state of affairs belongs to the possible states of affairs of the modelling. Otherwise the modelling is **false** in it.
- **symbols** are strings, (formal) **values** are mathematical objects (atomic objects, numbers, sets, relations, functions). Symbols and strings are mathematical objects too and they will sometimes be used as values. But in general, mathematical objects are not symbols or strings. E.g., a number is not a string; we distinguish between the number 5 and the symbol 5. A symbol for a number, e.g., 5, is called a **numeral**. We distinguished between **constants** (with a fixed value), **ontology symbols** and **quantified variables)**.
- **informal interpretation** $\mathcal{I}$: we use symbols to refer to objects, sets, relationships, functions, types in the domain. An informal interpretation $\mathcal{I}$ specifies the interpretation of mathematical symbols as informal concepts in the application domain. The informal interpretation captures the abstraction relation between mathematical symbols and objects and the informal domain of application. Applying an informal interpretation $\mathcal{I}$ to a mathematical statement returns a statement about the application domain. Applying an informal interpretation $\mathcal{I}$ to a mathematical structure $\mathfrak{A}$ returns a state of affairs of the application domain. [8]

- **structure**: an assignment of mathematical values to symbols. They serve as mathematical abstractions of states of affairs. Mathematical objects serve as abstractions of objects in the informal domain. A structure is sometimes called an *interpretation*.
- **a model** of a theory: a structure that satisfies the theory. Given an informal interpretation $\mathcal{I}$, the models of a theory are abstractions of possible states of affairs.
- Derived semantical relations.
  - Two propositions are **equivalent** if they are true in the same states of affairs.
  - One proposition **entails** a second proposition if each state of affairs satisfying the first also satisfies the second.
  - A proposition is **consistent** if it is true in at least one state of affairs.
  - A proposition is **inconsistent/contradictory** if false in every state of affairs.
- Given an informal interpretation $\mathcal{I}$, a formal scientific theory is **correct** if the models of the theory correspond to possible states of affairs of the application domain and vice versa.
- **Possible world analysis** is a technique to analyze the meaning of statements by showing states of affairs where they are true or where they are false.

1.2 The logic $\mathcal{L}in$

The language that formal scientists use in scientific text is a mixture of natural language and mathematical expressions. E.g., the statement "*the acceleration of b_i is proportional to the mass of b_j ($i \neq j$) and inverse proportional to the square of the distance.*" is a natural language statement. Nevertheless, this is a precise statement to the extent that a competent physicist would be able to reconstruct the corresponding equations from it.

In this section, we define the notion of a formal *modelling logic* and we define a toy example of one. This and the next sections serve (1) to specify the blue print for logics in which the modelling methodology of formal empirical science can be applied, (2) to formally define, once and for all, the important semantic concepts such as consistency, entailment, equivalence, (3) to create awareness of the range of computational problems that -in principle- can be solved using a modelling, (4) to define the concept of inference which is a domain independent computational problem (and problem solving method) that takes modellings as input.

Definition 1.2.1. A *modelling logic* is a trinity of:
(1) a formal definition of syntax (e.g., using an inductive definition or a grammar in Bachus Naur form),
(2) a formal model theory defining a formal notion of structure and model, and a satisfaction relation between structures and boolean expressions, and
(3) an *informal semantics*, an informal theory that for arbitrary informal interpretation $\mathcal{I}$ of a set Σ of symbols into some application domain, explains the meaning of the expressions of the logic in the application domain.

A potential problem with a modelling logic is what I will call the **semantical mismatch problem**: that formal and informal semantics do not match. There is a mismatch if we can find a logical sentence ψ, informal semantics $\mathcal{I}$ and a structure $\mathfrak{A}$ such that the truth value of ψ in $\mathfrak{A}$ disagrees with the proposition $\mathcal{I}(\psi)$ in $\mathcal{I}(\mathfrak{A})$. That is: the truth value of ψ in $\mathfrak{A}$ disagrees with the truth value of the informal semantics of ψ under $\mathcal{I}$ in the state of affairs that is abstractly represented as $\mathfrak{A}$ under $\mathcal{I}$.

We now introduce the (toy) modelling logic $\mathcal{L}in$. The concepts that we define for it such as structure, model, derived semantical concepts, inference, etc., are the same for other modelling logics in this course.

Syntax of $\mathcal{L}in$. The symbols used in expressions of $\mathcal{L}in$ are the following:

- The logical symbols:
  - An infinite set of *numerals*: symbols of the form $\ldots, -\mathfrak{2}, -\mathfrak{1}, \mathfrak{0}, \mathfrak{1}, \mathfrak{2}, \ldots$.
  - Binary arithmetic function symbols $+, \times$;
  - Binary comparison symbols: $=, <, >, \leq, \geq, \neq$.
- Ontology symbols, a.k.a. variables, often denoted as $x, y, \ldots$.

All logical symbols have a meaning fixed by the logic and have a fixed value which you have already guessed: they are the same as in mathematical text. The value of the numeral $\mathfrak{a}$ is the number a. The value of each comparison symbol is a subset of $\mathbb{N}^2$. We denote the value of a constant c as $c^{\mathcal{L}in}$. E.g., $\mathfrak{5}^{\mathcal{L}in}$ is the number 5 and $\neq^{\mathcal{L}in}$ is $\{(n, m) \in \mathbb{N}^2 | n \neq m\}$.

The *ontology symbols* are the variables. These are the symbols that a modeller can introduce as abstractions of (properties of) objects in the domain of discourse.

Below, we refer sometimes to ontology symbols as *variables*, following the standard terminology of this type of equations in mathematical modelling. In $\mathcal{L}in$, there can be no confusion with quantified variables, since $\mathcal{L}in$ has no quantificatio

Definition 1.2.2. A *vocabulary* Σ is a (finite or infinite) set of ontology symbols.

A *$\mathcal{L}in$-term over* Σ is a finite string t of the form

$$\mathfrak{a}_1 \times x_1 + \cdots + \mathfrak{a}_n \times x_n$$

where $\mathfrak{a}_1, \ldots \mathfrak{a}_n$ are numerals, $x_1, \ldots, x_n$ variable symbols.

A *$\mathcal{L}in$-sentence over* Σ is a finite string of the form

$$t \sim \mathfrak{a}$$

where t is a $\mathcal{L}in$-term over Σ, $\mathfrak{a}$ a numeral and $\sim$ an element of $\{=, <, >, \leq, \geq, \neq\}$.

A *$\mathcal{L}in$-expression over* Σ is a $\mathcal{L}in$-term or sentence over Σ.

A *$\mathcal{L}in$-theory* T over Σ is a finite or infinite set of $\mathcal{L}in$-sentences over Σ.

Formal semantics. Given a vocabulary Σ, a Σ-*structure* $\mathfrak{A}$ is an assignment of integer numbers to the symbols of Σ. I.e., it is a function $\Sigma \to \mathbb{N}$. The *value* of a symbol x in $\mathfrak{A}$, also called the *interpretation* of x in $\mathfrak{A}$, is denoted $x^{\mathfrak{A}}$.

Notation: the structure assigning 5 to x and -12 to y is denoted as $[x : 5; y : -12]$.

Definition 1.2.3. The *value* $t^{\mathfrak{A}}$ of a $\mathcal{L}in$-term t of the form $\mathfrak{a}_1 \times x_1 + \cdots + \mathfrak{a}_n \times x_n$ over Σ is the integer number $a_1 \times x_1^{\mathfrak{A}} + \cdots + a_n \times x_n^{\mathfrak{A}}$. This is also called the *interpretation of* t *in* $\mathfrak{A}$.

For any $\mathcal{L}in$-sentence e of the form $t \sim \mathfrak{a}$, we say that e is *true* in $\mathfrak{A}$ if the tuple $(t^{\mathfrak{A}}, \mathfrak{a}^{\mathcal{L}in})$ belongs to $\sim^{\mathcal{L}in}$. Otherwise, we say that e is false in $\mathfrak{A}$. Notation: if e is true in $\mathfrak{A}$, we denote this as $\mathfrak{A} \models e$. If e is false in $\mathfrak{A}$, we denote this as $\mathfrak{A} \not\models e$.

As such $\models$ is a binary relation between structures and sentences. We call it the **satisfaction relation** of $\mathcal{L}in$.

A theory T is true in $\mathfrak{A}$ (notation $\mathfrak{A} \models T$) if every sentence of T is true in $\mathfrak{A}$. Otherwise T is false in $\mathfrak{A}$.

If a sentence e (or theory T) is true in $\mathfrak{A}$, we say that $\mathfrak{A}$ *satisfies* e (or T) and that $\mathfrak{A}$ is a *model* of e (or T).

Given a sentence e, we define the truth value $e^{\mathfrak{A}}$ of e in $\mathfrak{A}$ to be $\mathbf{t}$ (true) if $\mathfrak{A} \models e$; otherwise $e^{\mathfrak{A}} = \mathbf{f}$ (false).

Informal semantics. Given some informal interpretation of variables, the informal semantics of a $\mathcal{L}in$-sentence reads as a numerical expression about the interpretations of the variables. We illustrate this in Example 1.2.1.

Example 1.2.1. Consider the following puzzle. *"I am one year older than twice the age of my son. The sum of our ages is between 70 and 80. What are our ages?"* This puzzle contains information and a problem. We introduce vocabulary $\Sigma = \{x, y\}$ and the informal interpretation $\mathcal{I}(x)$: my age and $\mathcal{I}(y)$: my sons age. For a structure, e.g., $\mathfrak{A} = [x : 100, y = 1]$, $\mathcal{I}(\mathfrak{A})$ is the (non-actual) state of affairs where I am hundred years old, and my son is 1 year old.

The modelling of the information in $\mathcal{L}in$ is the following theory T_{age}:

$$\begin{aligned} \mathfrak{1} \times x + -\mathfrak{2} \times y &= \mathfrak{1} \\ \mathfrak{1} \times x + \mathfrak{1} \times y &\geq \mathfrak{70} \\ \mathfrak{1} \times x + \mathfrak{1} \times y &\leq \mathfrak{80} \end{aligned}$$

Given the informal interpretation $\mathcal{I}$, for each of these equations e, $\mathcal{I}(e)$ is a precise proposition about our ages. E.g., the first one expresses that my age minus two times my sons age is 1; i.e., that I am one year older than twice the age of my son. The three sentences of this theory corresponds to the two pieces of the puzzle.

The structures satisfying T_{age} are solutions of the equations. One can verify that there are four models with my age ranging around 50, and that of my son around 25. Thus, the puzzle has four models which correspond to four possible states of affairs. ■

Does the semantical mismath occur in $\mathcal{L}in$? If so, we can find a theory T, a structure $\mathfrak{A}$, an informal semantics $\mathcal{I}$ such that $\mathfrak{A} \models T$ but $\mathcal{I}(T)$ is false in the state of affairs $\mathcal{I}(\mathfrak{A})$. For example, imagine there is a structure $\mathfrak{A}$ satisfying the theory T_{age} of the puzzle, but it specifies ages for me and my son that are not solutions of the puzzle.

But, such triples $T, \mathfrak{A}, \mathcal{I}$ do not exist: the way we interpret $\mathcal{L}in$-sentences e as mathematical equations about numerical entities in the application domain corresponds to the way $\mathfrak{A} \models e$ is defined. So, yes, formal and informal semantics of $\mathcal{L}in$ match. Thus, we can safely translate linear equations to expressions in $\mathcal{L}in$.

The following exercise shows that even a smallest mistake in the definition of the formal semantics would lead to a semantic mismatch problem, and that possible world analysis would reveal it.

Example 1.2.2. Assume we keep the informal interpretation of $<$ (as "strictly less than") but formally define $<^{\mathcal{L}in} = \{(5,5), (n,m) | n < m\}$. Now, $<^{\mathcal{L}in}$ contains one erroneous tuple $(5,5)$. This is a small mistake, given that the relation has infinitely many elements. Show a theory T, a structure $\mathfrak{A}$, an informal interpretation $\mathcal{I}$ showing the semantic mismatch.

This is a trivial exercise. Take theory $T = \{\mathtt{1} \times x < \mathtt{5}\}$. Take structure $\mathfrak{A} = [x : 5]$, take $\mathcal{I}(x)$ to be my sons age. Then $\mathfrak{A} \models T$ since $(5,5) \in <^{\mathcal{L}in}$, while $\mathcal{I}(T)$ is false in state of affairs $\mathcal{I}(I)$: " my son is strictly less than 5 years" is false in a state of affairs where he is 5.

In the next chapter, we will see that in FO, the issue of semantic mismatches is far from trivial.

Derived semantical concepts: formal definitions The derived semantical concepts of entailment, equivalence, consistency, contradiction, are all as explained in the previous section, in terms of the satisfaction relation $\models$.

The definitions below give a formal definition of these concepts in the context of an arbitrary logic that possesses a satisfaction relation $\models$. They will apply to all logics that we will see in this course.

Definition 1.2.4. Let Σ be a vocabulary, T, T' be theories or sentences over Σ.

T is **satisfiable** or **consistent** if at least one structure satisfies T. T is **contradictory** or **unsatisfiable** if there is no structure that satisfies T.

T is **tautological** or **logically valid** (notation $\models T$) if T is satisfied in every Σ-structure.

T is **logically equivalent** to T' (notation $T \equiv T'$) if T, T' are true in the same structures. Or equivalently, if for each structure $\mathfrak{A}$ over Σ, $T^{\mathfrak{A}} = T'^{\mathfrak{A}}$.

T **logically entails (or logically implies)** T' (notation $T \models T'$) if every structure $\mathfrak{A}$ that satisfies T satisfies T'.

T contains **complete knowledge** if it has only one model. Such a theory is called **categorical**.

Notice the overloading of the symbol $\models$. The binary relation $\mathfrak{A} \models T$ is the satisfaction relation. The binary relation $T \models T'$ (between theories) is the entailment relation. The unary relation $\models T$ is the validity property.

Categorical theories provide maximal consistent information. They characterize a unique state of affairs.

The semantic concepts defined in Definition 1.2.4 exist in formal sciences, and their understanding is formalized in this definition.

Example 1.2.3. In $\mathcal{L}in$ we have the following formally verifiable properties:

- The sentence $2 \times x > 4$ is satisfiable. The structure $[x : 3]$ is a model of it.
- The sentence $1 \times x + -1 \times x = 0$ is a tautology (of is logically valid). Equivalently, $1 \times x + -1 \times x \neq 0$ is contradictory.
- The sentence $2 \times x > 4$ is equivalent to $1 \times x \geq 3$.
- The sentence $2 \times x > 4$ entails $1 \times x > -10$.
- The theory $\{1 \times x = 23, 1 \times y = 54\}$ is categorical. The theory T_{age} of the age puzzle is not categorical.

■

The semantical concepts are strongly interrelated. E.g., T is logically valid iff $\neg T$ is contradictory iff $\neg T$ is not satisfiable. T is logically equivalent to T' iff T entails T' and T' entails T. The following property is important in theorem proving, since it reduces an entailment question to a simpler satisfiability question.

Proposition 1.2.1. *T logically entails φ iff $T \cup \{\neg\varphi\}$ is unsatisfiable.*

Proof. Let Σ be the vocabulary of T and φ. T logically entails φ
iff every Σ-structure $\mathfrak{A}$ that satisfies T, satisfies φ
iff there is no Σ-structure $\mathfrak{A}$ such that $\mathfrak{A} \models T$ and $\mathfrak{A} \not\models \varphi$
iff there is no structure $\mathfrak{A}$ satisfying T and $\neg\varphi$
iff $T \cup \{\neg\varphi\}$ is unsatisfiable. ■

1.3 Computational problems and inference

A theory in $\mathcal{L}in$ is not a program but it represents information. Information determines the solutions to potentially many problems. Below, a connection between computational problem solving and information is sketched.

Definition 1.3.1. A *computational problem* is a function $\mathcal{M}$ between two types of (finitely encodable) mathematical objects:

- input objects are elements of some domain $In(\mathcal{M})$

- output objects are elements of some domain $Out(\mathcal{M})$.

An *instance* of the problem $\mathcal{M}$ is given by a specific input object $i \in In(\mathcal{M})$. Its solution is $\mathcal{M}(i)$.

A computational problem $\mathcal{M}$ is *computable* if there is a program that for every instance i, returns $\mathcal{M}(i)$ and terminates.

A computational problem $\mathcal{M}$ is a *decision problem* if $\mathcal{M}$ is a Boolean function.

A decision problem $\mathcal{M}$ is *decidable* if it is computable. It is *semi-decidable* if a program exists that for each input i returns $\mathbf{t}$ if $\mathcal{M}(i) = \mathbf{t}$ and otherwise, returns $\mathbf{f}$ or does not terminate.

[9] Computational problems may be incomputable (such as the halting problem) or computable. Decision problems can be undecidable, semi-decidable or decidable. Decidable decision problems can be further partitioned in complexity classes, e.g., P, NP, Exp, NExp, ...

Solving computational problems using information Information allows us to solve (computational) problems. Typically, not just one particular type of problem, but a range of problems. We discussed this already for Newtons gravitation theory. Below, a few problems in the age puzzle are specified that can be solved using the information in the age puzzle theory T_{age}.

Example 1.3.1 (Example 1.2.1, continued)**.** Below is a list of problems that can be answered using the information in T_{age}. Importantly, take notice how the solution is defined in terms of the semantics of T_{age}.

1. Decide if it is possible that I am 46 and my son 22. The answer is yes iff $[x : 46; y : 22] \models T_{age}$.
2. Decide if the puzzle is consistent/satisfiable. The answer is yes iff T_{age} has a model.
3. Return all possible solutions to the puzzle. The list of possible solutions is the list of models of T_{age}.
4. Decide if the puzzle entails that my son is adult. The answer is yes iff in each model $\mathfrak{A}$ of T_{age}, $\mathfrak{A} \models y \geq 18$. Equivalently, iff T_{age} entails $y \geq 18$.
5. Given the additional clue that the right solution is the one where my age is maximal, determine our ages. Return the model $\mathfrak{A}$ of T_{age} where $x^{\mathfrak{A}}$ is maximal, i.e., such that for any other model $\mathfrak{A}'$ of T_{age}, it holds that $x^{\mathfrak{A}} \geq x^{\mathfrak{A}'}$.
6. Compute the possible ages of my son. Return the set $\{y^{\mathfrak{A}} \mid \mathfrak{A} \models T_{age}\}$.
7. Explain why $[x : 53; y : 26]$ is a solution; or why my son is adult, or why 34 is not a possible age of my son, ...

Notice that since the answers can be defined in terms of the models of T_{age}, the answer of each problem would be the same if we replace T_{age} by an equivalent theory. ■

[9] In this definition, the programming language is not specified. According to **Church's hypothesis**, all sufficiently rich programming languages are equally expressive.

Exercise 1.3.1. *Can you find other sorts of problems that can be solved with the information of the puzzle?*

For such a tiny puzzle, the list of problems solvable using the information in the age puzzle is surprisingly long and heterogenous. Each of the problems stated above can be generalized to a generic problem in which a logic theory T itself is one of the input parameters. In doing so, each of these problems is generalized to a type of **inference**.

But what is inference? Below we define it as a sort of computation based on the information expressed in an input theory.

Definition 1.3.2. An *inference problem* of a modelling logic $\mathcal{L}$ is a computational problem $\mathcal{M}$ that takes a theory (or logical proposition) of $\mathcal{L}$ as one of its input arguments such that if T and T' are logically equivalent theories of $\mathcal{L}$, then the output $\mathcal{M}(T, \dots)$ and $\mathcal{M}(T', \dots)$ is the same or is equivalent.

The idea of inference is that only the information content of the input theory matters, not the precise syntactical form. Thus, computing the number of symbols of an input theory T is *not* an inference problem. Theories that are equivalent have the same information content, hence substituting one for another in the input should not have an impact on the output.

Below, we define some inference problems. This list is not exhaustive. The definitions hold for all modelling logics with a model-theoretic semantics.

Definition 1.3.3. Below, we define a number of forms of inference.

- The *evaluation* inference problem $\mathcal{M}_{eval}$:
  - Input: a Σ-structure $\mathfrak{A}$, and term or expression or theory e over Σ
  - Output: $e^{\mathfrak{A}}$.

  If e is a Boolean expression, the outcome is $\mathbf{t}$ ("true") if e is satisfied in $\mathfrak{A}$, and $\mathbf{f}$ otherwise. This case is often called *model checking inference*.
- The *satisfiability checking* problem $\mathcal{M}_{sat}$:
  - Input: theory T;
  - Output: $\mathbf{t}$ if T is satisfiable, $\mathbf{f}$ otherwise.

  Recall T is satisfiable/consistent if some structure (interpreting its variables) satisfies T.
- The *model generation* inference problem $\mathcal{M}_{modgen}$:
  - Input: theory T;
  - Output: "Unsat" if T is unsatisfiable, otherwise a model $\mathfrak{A}$ of T.
- The *deduction* inference problem $\mathcal{M}_{ded}$:
  - Input: theory T, boolean expression e;

  - Output: **t** if T logically entails e, **f** otherwise.

- The *validity checking* inference problem $\mathcal{M}_{valid}$:
  - Input: theory T;
  - Output: **t** if T is logically valid, **f** otherwise.

- The *possible values* inference problem $\mathcal{M}_{posval}$:
  - Input: theory T, a term t
  - Output: the set of all values $t^{\mathfrak{A}}$ of t in all models $\mathfrak{A}$ of T.

- The *minimization* problem $\mathcal{M}_{minim}$:
  - Input: theory T, arithmetical term t
  - Output: "Unsat" if T is unsatisfiable, otherwise a model $\mathfrak{A}$ of T such that for any model $\mathfrak{A}'$ of T, $t^{\mathfrak{A}} \leq t^{\mathfrak{A}'}$.

Proposition 1.3.1. *Each problem defined in Definition 1.3.3 is an inference problem.*

Proof. When replacing the theory T or the expression e in the input by an equivalent theory T' or formula e', we need to prove that the output does not change. This is easy to verify. E.g., take the deductive inference problem $\mathcal{M}_{ded}$, substitute an equivalent pair of theory and formula T', e' for T, e. Then T' logically entails e' iff every model of T' is a model of e' iff every model of T is a model of e. Hence, $\mathcal{M}_{ded}(T, e) = \mathcal{M}_{ded}(T', e')$. ■

The above inference problems are well-defined computational problems in every modelling logic.

Example 1.3.2. The question if my son is adult can be formulated as a *problem instance* of the deduction inference problem. Namely, as the instance (T_{age}, e) where T_{age} is the age theory and e is the formula $\mathfrak{1} \times y \geq \mathfrak{18}$.

Exercise 1.3.2. *Classify the different problems in Example 1.3.1 as instances of the inference problems in Definition 1.3.3. Which ones cannot be classified as a form of inference?*

The discussion above points to the potential of formal modellings for problem solving: the possibility of reusing a theory of T to solve a range of different types of problems in the application domain. This sort of reuse cannot be achieved with standard programs. Indeed, programs tend to be problem-specific: they are implemented to solve a specific problem.

However, there is no free lunch in the world. To solve a specific problem instance applying a specific sort of inference on a domain theory T, we need an efficient inference program that takes as input T and reasons on its information. To develop such programs is often the million dollar problem. For some forms of inference, efficient inference programs exist (one could say that database systems efficiently solve evaluation inference, since a query is a special term and a database DB is a special structure). But for other forms of inference arising in a particular

application domain, the domain specific program written by a human programmer is often much more efficient than state of the art inference engines.

One of the problems is that the computational properties of different sorts of inference may vary enormously. E.g., for the logic FO (First Order Logic) seen in the next chapter, the model checking inference problem of verifying whether a formula is satisfied in a finite structure is polynomial in the size of the structure, but the satisfiability inference problem for a formula is undecidable and the deduction inference problem is semi-decidable. [10] Thus, evaluating an FO proposition in a structure may take a fraction of a second, while verifying that the same proposition is satisfiable or valid may never terminate. The complexity of computational inference problems varies widely, from efficient to undecidable. It means that we cannot associate a complexity class to a modelling logic (unless we fix the sort of inference). This is life, this is the nature of knowledge, we have to accept it.

1.4 Modelling in the current state of the art

Software systems are becoming more and more complex. Building complex software starts with a design phase where detailed specifications of different aspects of the system are made: modelling domain knowledge (e.g., the work scheduling domain, or the dynamic system), functional and non-functional requirements, deployment, etc).

Modelling is necessary to build *correct* systems. Currently, most software contain bugs. Most of us have learned to live with them, but there are applications where bugs are unacceptable and building *correct* software is crucial. Such applications are sometimes called *mission critical*. They are applications where bugs can be extremely costly, in terms of money, resources, or human lives. A few costly bugs taken from `https://en.wikipedia.org/wiki/List_of_software_bugs`:

- In the 80ties, several cancer patients died due to overdoses of radiation resulting from a race condition between concurrent tasks in the Therac-25 software.
- On 4 June 1996, the Ariane 5 crashed after 40 seconds on its maiden flight. It was caused by a faulty software exception routine resulting from an overflow while converting a 64-bit floating point to a 16-bit integer.
- On 23 September 1999, the Mars Climate Orbiter crashed on Mars due to the fact that some modules expressed propulsion power in pound-seconds, while other modules interpreted these numbers in newton-seconds.
- Millenium bug problems.
- The Pentium FDIV bug in the Intel P5 Pentium floating point unit discovered in 1994 made Intel loose $475 million dollars.
- The Samsung Note 7 fiasco was a problem with exploding batteries in Note 7 smartphones. Experts speculate that the problem was a bug in the software that monitors battery charge. It costed Samsung nearly $17 billion.

The common solution to avoid bugs is testing, testing, testing, but this does not guarantee correctness of software in all eventualities.

[10] This is known as the well-known undecidability and semi-decidability of FO.

When correctness is essential, there is no other way than through formal modelling and verification. At present, it may still be very costly. On the other hand, using formal verification can also save money. It is well-known that how earlier a bug is found, how cheaper it is to solve it. According to the Systems Sciences Institute at IBM it costs 6x more to fix a bug found during implementation than during design, and 15x more during the testing phase. The cost of solving a bug in operational software may be exponentially much higher. [11]

As said before, using formal modelling and verification may still be very expensive and time-consuming. However, in certain types of applications, formal modelling can also be very time saving. E.g., a company that uses a constraint solver to solve a complex scheduling or configuration problem using a declarative specification of the configuration domain, probably has saved much time and money by not building its own (domain-specific) scheduling program.

How to use Modellings? Currently, modellings serve three main purposes in software engineering: (1) modellings are useful for precise documentation, (2) modellings can be used to develop provably correct systems designs, (3) modellings can be used by computers to solve problems and perform tasks.

As for (1), a modelling provides precise *documentation* of the modeled domain. When complex systems are built, precise documentation is crucial for maintaining them. This is not only the case in software, but also in hardware. E.g., there is a rumor that after the space shuttle program collapsed due to accidents with the Challenger and Columbia, NASA could not rebuild the reliable Saturn 5 rockets used in the Apollo program because documentation had not been maintained. In the context of software, the Unified Modelling Language (UML) is mainly concerned with (1). UML consists of different formal languages to describe different aspects of a software system.

The use of (2) goes one step further and uses the formal modelling(s) of the system for analysis of its correctness properties, and perhaps for development of formally verified computer programs. Different systems follow this road to different length.

For example, the Alloy community [12] uses the formal specification language Alloy to specify system components, and run a range of so called *light weight* verifications on it. We discuss this in Chapter **??**. This way, bugs can be removed during the design phase and the confidence of the designer grows that eventually the program implementing the formally specified design, will behave correctly. However, the Alloy specifications built in this stage only serve to analyze the specification; they are not used to implement the software. The implementation restarts from scratch.

A much stronger way of using the formal specification is found in the B-method, and its descendant Event-B which will be studied in Chapter 5. An early illustration of this approach is the control system of Metro 14 in Paris, a fully automatic metro system operational since 1998. The control software was developed by Siemens. It was compiled from a sequence of more and more refined formal specifications, each of which had been formally checked for correctness with respect to the previous specification. The result was a program of 86.000 lines of Ada/C/Java code. In comparison, the specification in the B-method was 110.000 lines in the B language, showing that its development was extremely labor intensive. However, this system achieved something exceptional: during the first ten years of operation, until it was upgraded to a new

[11] `https://www.synopsys.com/blogs/software-security/cost-to-fix-bugs-during-each-sdlc-phase/`
[12] `https://en.wikipedia.org/wiki/Alloy_(specification_language)`

version of the hardware, not a single bug was found. [13] In this course, we study event-B, a descendant of B and the refinement methodology that it supports.

Use (3) of modelling adds one extra step: problems are directly solved using the modelling, by applying generic (domain independent) inference programs to them. No more (domain specific) programming is required. Here we are close to the several *declarative programming paradigms*: logic and functional programming, Constraint Programming, databases and several other young and dynamic fields such as Answer Set Programming and the Semantic Web. Other approaches following (3) are executable declarative work flow languages such as LogicBlox.[14] Theories in LogicBlox specify workflow processes consisting of interactions between system, users, databases and other actors. These logic theories can be used for various purposes, including the most obvious one: *executing* the specified process in the run-time environment. Chapter 7 presents a view on this way of using modellings.

In this course, we provide a overview of principles of a few different approaches in (2) and (3). Of course, it is not possible to introduce all these languages and systems. However, it is possible to explain many principles in terms of the (few) languages used in this course.

1.5 Mathematical preliminaries for this course

The Boolean values are $\mathbf{t}$ and $\mathbf{f}$. They stand for "true" and "false". The set of Boolean values is denoted $\mathbb{B} = \{\mathbf{t}, \mathbf{f}\}$.

We distinguish between several sorts of numbers: natural (non-negative) numbers $\mathbb{N}$, integers $\mathbb{Z}$, rationals $\mathbb{Q}$ and reals $\mathbb{R}$. (Complex numbers do not show up in this course.)

There are many sorts of mathematical objects: sets, numbers, tuples, relations, functions, matrices, etc. However, in the eye of a mathematician, (almost) any mathematical object is ultimately a *set*. E.g., a matrix can be viewed as a function mapping tuples of indices $(i_1, \ldots, i_n)$ to the content of the cell indexed by $(i_1, \ldots, i_n)$. In turn, any function and relation can be viewed as a set, as explained below. Even tuples and numbers have been encoded in terms of sets (but we will not go so far).

A *set* is a collection of objects. The identity of a set is determined by its elements. Hence, two sets are equal if and only if they contain the same elements. Some notations:

- $\emptyset$: the empty set, the set that has no elements
- $x \in X$: x is an element of X
- $X \subseteq Y$: X is a subset of Y; i.e., all elements of X are element of Y.
- $\mathcal{P}(X)$ is the power set of X, the set $\{Y \mid Y \subseteq X\}$.

Mathematical properties of sets were studied in the classical logic theory of Zermelo and Fraenkel (ZFC). Since almost every mathematical object can be viewed as a set, these axioms are considered to be the basic axioms of mathematical reasoning. It is assumed that every theorem that was ever proven in mathematics can be rephrased as a theorem in ZFC.

[13] `http://deploy-eprints.ecs.soton.ac.uk/8/1/fm_sc_rs_v2.pdf`
[14] `http://www.logicblox.com/`

An n-tuple $(a_1, \ldots, a_n)$ is a sequence of n elements.

The Cartesian product $D_1 \times \cdots \times D_n$ of sets $D_1, \ldots, D_n$ is the set $\{(a_1, \ldots, a_n) \mid a_1 \in D_1, \ldots, a_n \in D_n\}$. If $D_1 = \cdots = D_n$ we denote this as D^n.

A relation R with domain $D_1 \times \cdots \times D_n$ is a subset of $D_1 \times \cdots \times D_n$. We call n the arity of R. An n-ary relation R on domain D is a subset of D^n. Thus, a relation is a set of n-tuples, and a set is a 1-ary relation.

A function f with domain D and range S is a set of 2-tuples $(x, y) \in D \times S$ such that, for every $d \in D$ there exists a unique element $y \in S$ such that $(x, y) \in f$. We denote y as $f(x)$. The set of functions with domain D and range S is denoted $D \to S$. We write $f : D \to S$ to denote that $f \in (D \to S)$. To distinguish with partial functions, we sometimes call a function a *total* function.

A partial function f with domain D and range S is defined similarly: for every $d \in D$ there exists *at most* one element $y \in S$ such that $(x, y) \in f$. In this case, $f(x)$ may not exist. The set of partial functions from D to S is denoted $D \rightharpoonup S$. It holds that $D \rightharpoonup S = \cup_{C \subseteq D}(C \to S)$. We write $f : D \rightharpoonup S$ to denote that f is a partial function from D to S.

Given a domain D, the characteristic function $\chi_A : D \to \mathbb{B}$ of a subset $A \subseteq D$ maps $x \in A$ to $\mathbf{t}$ and $x \in D \setminus A$ to $\mathbf{f}$. Vice versa, the *graph* $\mathcal{G}_f$ of a (partial) function $f : D_1 \times \cdots \times D_n \to S$ is the relation $\{(a_1, \ldots, a_n, y) \in D_1 \times \cdots \times D_n \times S \mid f(a_1, \ldots, a_n) = y\}$. There is a one-to-one correspondence between relations and their characteristic functions, and vice versa between (partial) functions and their graphs.

A bijection $f : D \to S$ is a total function such that for every $y \in S$, there exists a unique $x \in D$ such that $f(x) = y$. In mathematical notation, if $\forall y \in S : \exists! x \in D : f(x) = y$. Thus, a bijection defines a one-to-one correspondence between elements of D and of S.

Bijections are the mathematical means to establish that two sets have the same properties. E.g., if there is a bijection from X to Y, then both have the same cardinality. It is used to mathematically define the idea of *isomorphism*. See next chapter.

An injection $f : D \to S$ is a total function such that for every $y \in S$, there exists at most one $x \in D$ such that $f(x) = y$. In mathematical notation, if $\forall y \in S : \#\{x \in D \mid f(x) = y\} \leq 1$ (the number of x that map to y is 0 or 1).

An surjection $f : D \to S$ is a total function such that for every $y \in S$, there exists at least one $x \in D$ such that $f(x) = y$. In mathematical notation, if $\forall y \in S : \exists x \in D : f(x) = y$.

One can see that a total function is a bijection iff it is an injection and a surjection.

A binary relation G on domain D is *reflexive* if for all $d \in D$, $(d, d) \in G$.

G is *symmetric* if for all $(d, e) \in G$ it holds that $(e, d) \in G$.

G is *asymmetric* if $(d, e) \in G$ and $(e, d) \in G$ entails that $d = e$. That is, if $d \neq e \in D$ then it is not the case that $(d, e), (e, d) \in G$.

G is *transitive* if for all $(d, e), (e, f) \in G$, it holds that $(d, f) \in G$.

A *graph* G on domain E is a binary relation on E. We often call the elements of E the vertices or nodes. We call a pair $(x_i, x_{i+1}) \in G$ an *edge* of G.

An undirected graph is a symmetric graph, one such that if $(x, y) \in G$ then $(y, x) \in G$. A path

in G is a finite or infinite sequence $\langle x_0, x_1, \ldots, \rangle$ such that each pair (x_i, x_{i+1}) in this sequence is an edge in G.

For all $x, y \in E$, we say that y is *reachable* from x in G if there exists a finite path in G with start x and end y.

A graph is *connected* if for every pair of nodes, there is a path from one to the other.

1.6 Important for the exam

```
Questions for the exam: definitions of the derived semantics concepts
 (tautology, equivalence, entailment, ...); definitions of forms of
 inference.

There will be no specific questions on how formal science works or on  the logic Lin
 but, as stated  above, there might be questions of formal definitions of semantical
 concepts and forms of inference problems that were defined in the
 context of this logic for the first time, but that hold for all
 logics with a satisfisfaction relation.

You should understand the fundamental concepts that were derived from
 formal sciences since these concepts are all applied to modelling and
 modelling logics. All of the following concepts are of importance:

- problem domain - application domain - domain of discourse

- states of affairs of the application domain

- symbols - values - structures as assignments of values to symbols
- informal  interpretation of symbols in the domain
- structures as mathematical abstractions of states of affairs of the domain

- propositions - truth/satisfaction relation between structures and propositions
- informal semantics of propositions

- a modelling logic
- formal semantics
- the satisfaction relation
- satisfiability
- validity (tautology)
- entailment
- equivalence

Using information for solving problems:
- computational  problems
- inference problems:
  evaluation,  model checking,
  satisfibility ,
  deduction,
  model generation,
```

Chapter 2

Predicate Logic and extensions

2.1 Short history

The old greeks invented *mathematical reasoning*. *Aristoteles* (384-322 v.C.) observed that there were patterns in correct mathematical reasoning, and he studied some of them in his *syllogisms*. An example is the Barbara:

$$\frac{\begin{array}{c}\text{All men are mortal.}\\ \text{All Greeks are men.}\end{array}}{\text{All Greeks are mortal.}}$$

He started the scientific study of this form of reasoning: deductive reasoning – a great exploit.

Philosophers and mathematicians have been searching for more than 2000 years to further formalize and extend these patterns. Only in the 19th century substantial progress was made over Aristoteles.

Gottfried Wilhelm Leibniz (1646-1716) was a genius German philosopher, mathematician and diplomat. Leibniz's greatest work is the differential and integration calculus, which he developed (largely) independently of Newton. But he was active in many other domains, including logic. He was convinced that:

- Complex *thoughts* in our mind are composed from simpler thoughts, using mathematical operators analogous to arithmetical operators $+, \times, -, \ldots$.
- Once we make our thoughts explicit in terms of these operators, it will be possible to study the rules of correct reasoning using mathematics.

Leibniz phrased and partially formalized these ideas in letters but he did not publish his results. Only around 1840, his work was published and his ideas became broadly known and started to influence the development of logic. [1] Leibniz had a strong belief in logic and believed that it would allow to solve all diplomatic disagreements by rational reasoning. That may have been a little naive.

[1] https://plato.stanford.edu/entries/leibniz-logic-influence

George Boole (1815-1864) identified a number of these operators, such as $\wedge, \vee, \neg$ in a paper "*An Investigation of the Laws of Though (1854)*". In this paper, he developed *Boolean algebra*, work that laid the foundations for digital circuits. In logic, it is known as *propositional logic*. He was a brilliant, humble and decent man, who died at middle age and got fame only after his death. His death was due to superstition. In his time, some believed that a disease could be cured by repeating the cause of the disease. One day, he caught a cold during a walk through heavy rain. When he was in bed with high fever, his wife doused him with a bucket of cold water. He died of pneumonia.

Gottlob Frege extended this work by adding *quantification*, and by extending the set of inference rules of correct reasoning. His *Begriffschrift* (1879) counts as the first presentation of *classical logic*.

Kurt Gödel was probably the main logician of the first part of the 20st century, and resolved a number of fundamental questions regarding the foundation of mathematical reasoning. One result is his completeness theorem: that each proposition entailed by a classical logic theory, has a proof in one of the standard proof systems. Another famous result is his incompleteness theorem: that every enumerable/computable classical logic theory T of the natural numbers is necessarily incomplete in the sense that there are true propositions about the natural numbers that cannot be proven from T. Chapter 6 discusses these results (and some others).

Classical logic as a modelling language was a by-product of the study of deductive inference. Indeed, information is the raw material for any form of reasoning. To formally study reasoning, one needs a formal language to express information. Leibniz, Boole, Frege, Gödel and so many others not only contributed to deductive reasoning but also to the development of a language in which to express the resource for reasoning: the information. This led to development of the language of classical logic as a modelling language.

Classical logic is by no way an ideal modelling language. This chapter will propose a few very useful extensions. However, classical logic is a base language because its language constructs are of fundamental importance.

2.2 First-order Logic (FO)

In this chapter, we apply the ideas of Chapter 1 to build first order logic, also called predicate logic. We abbreviate this logic as FO (also FOL is commonly used). We also present a number of extensions of this logic.

2.2.1 Symbols, values and structures

According to FO, a state of affairs can be abstracted as a set D of atomic objects, and a designated set of relations and total functions on D. A number of ontology symbols, called the non-logical symbols, designate some of the atomic objects in D, the relations and functions.

Definition 2.2.1. A vocabulary is a set Σ of **non-logical symbols**. A non-logical symbol σ is either a function symbol or a predicate symbol. Each symbol σ has an arity n. A function symbol of arity 0 is called an object symbol. A predicate symbol of arity 0 is called propositional symbol.

Notation in this course: a function symbol is denoted as Sam/0:, PrimeMinister/0:, AgeOf/1:, +/2:, ... A predicate symbol is denoted as Sunny/0, Human/1, CourseResult/3, ... This specifies the symbol, its arity and in case of trailing ":" that it is a function symbol.

Non-logical symbols correspond to ontology symbols as introduced in the first chapter. A user introduces a non-logical symbol σ to represent a concept in the application domain. This concept is the informal interpretation of σ. E.g., in the context of a course scheduling application, the user may have introduced the 0-ary function symbol $LecturerOfLogic$ for expressing het concept of the lecturer of the course Logic, and the ternary predicate symbol $CourseResult/3$ for the relation between a student, a course for which he was examined and the score of the exam. The collection of chosen symbols and the concepts they stand for constitutes the informal interpetation $\mathcal{I}$ of Σ.

Notation in the IDP-language The IDP3 knowledge base system supports an extension of FO. In the language of this system, a vocabulary is described as in the following example.

Example 2.2.1. Take the vocabulary Σ={ 0/0:, S/1:, ST/0, P/1, Q/2 }. FO is untyped while IDP3 supports a typed version of FO as introduced in Section 2.3.1. We simulate the untyped universe by type D. The following example writes Σ in IDP3 style. To each declared symbol, I have added a comment explaining a proposed informal interpretation:

```
vocabulary V{
    type D
    0:D      // King Filip I of Belgium
    S(D):D   // S(x):  Father of x
    ST       // ST: it is sunny weather today
    P(D)     // P(x):  x is a liberal
    Q(D,D)   // Q(x,y):  x is a grandparent of y
    }
```

In this vocabulary, under the proposed $\mathcal{I}$, the term $S(0)$ represents the former king Albert II.■

Logical symbols The logic also provides a set of **logical symbols**. They are symbols with a meaning fixed by the logic. They serve as language constructs to build compound propositions from more primitive propositions (in the spirit of Leibniz). These symbols are well-known from mathematical texts where they are used as abbreviations. The table below presents them together with their informal interpretation.

$=$	equality, ... is equal to ...
$\wedge$	conjunction, ... and...
$\vee$	disjunction, ... or ...
$\neg$	negation, not ...
$\Rightarrow$	material implication, if ... then ...
$\Leftrightarrow$	equivalence, ... if and only if ...
$\exists$	existential quantor, there exists $\langle$ variable $\rangle$ such that ...
$\forall$	universal quantor, for all $\langle$ variable $\rangle$, ...

Table 2.1: Table of logical symbols and their informal interpretation

Finally, there is another type of symbol used in FO theories: quantified variables.

Definition 2.2.2. A *structure* $\mathfrak{A}$ of vocabulary Σ consists of
- a set $D_{\mathfrak{A}}$, called the *domain* or also, the *universe* of $\mathfrak{A}$;
- for each symbol $\sigma \in \Sigma$ an appropriate value denoted $\sigma^{\mathfrak{A}}$:
  - for an object symbol $C \in \Sigma$, a domain element $C^{\mathfrak{A}} \in D_{\mathfrak{A}}$;
  - for an n-ary function symbol $F \in \Sigma$, an n-ary (total) function $F^{\mathfrak{A}} : {D_{\mathfrak{A}}}^n \to D_{\mathfrak{A}}$;
  - for a propositional predicate symbol $P \in \Sigma$, a truth value $P^{\mathfrak{A}} \in \mathbb{B} = \{\mathbf{t}, \mathbf{f}\}$;
  - for an n-ary predicate symbol $P \in \Sigma$, an n-ary relation $P^{\mathfrak{A}} \subseteq {D_{\mathfrak{A}}}^n$;

Each object symbol C/0: $\in \Sigma$ is viewed as 0-ary function. Strictly speaking the value $C^{\mathfrak{A}}$ of a such a symbol should then be a 0-ary function mapping the unique 0-ary tuple () to some domain element. But we have identified $C^{\mathfrak{A}}$ with the domain element itself.

Likewise, each propositional symbol $P \in \Sigma$ is viewed as a 0-ary predicate symbol. Hence, strictly speaking, $P^{\mathfrak{A}}$ is 0-place relation. There are two 0-ary relations $\emptyset$ and $\{()\}$. These relations are identified with the boolean values, respectively $\mathbf{f}$ and $\mathbf{t}$. Thus, the value of a propositional symbol is $\mathbf{f}$ or $\mathbf{t}$.

Given an informal interpretation $\mathcal{I}$ of its vocabulary Σ, a structure is an abstraction of a state of affairs of the application domain, representing a collection of existing entities corresponding to $D_{\mathfrak{A}}$, and relations between them, and function defined on them.

Denoting structures Structures will be denoted in several styles.

Example 2.2.2. Take the vocabulary $\Sigma = \{O/0 :, S/1 :, ST/0, P/1, Q/2\}$. We define one FO structure $\mathfrak{A}$ for this vocabulary, and illustrate 3 different notations for it.

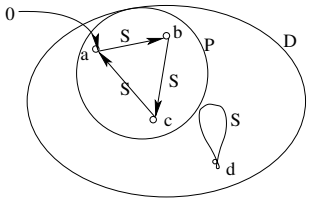
- Mathematical notation:
  - $D_{\mathfrak{A}} = \{a, b, c, d\}$
  - $O^{\mathfrak{A}} = a$
  - $S^{\mathfrak{A}} = \{(a, b), (b, c), (c, a), (d, d)\}$ representing a total function as a set of pairs;
  - $ST^{\mathfrak{A}} = \mathbf{t}$
  - $P^{\mathfrak{A}} = \{a, b, c\}$
  - $Q^{\mathfrak{A}} = \{(a, a), (a, b), (b, c)\}$

  Notice that a, b, c, d are domain elements. They are not symbols belonging to the vocabulary.

- Notation in IDP-syntax: the following defines the IDP3-structure `S` of vocabulary `V`:

```
structure S:V{
    D={ a; b; c; d}
    O=a
    S={a-> b; b->c; c->a; d->d}
    ST=true
    P={a; b; c}
    Q={(a,a); (a,b); (b,c)}
}
```

- We often use a graphical notation, but only for object symbols, and unary predicate and function symbols. E.g., the diagram below represents the above structure, restricted to these symbols. Values of other symbols cannot (easily) be represented in this graphical way.



Exercise 2.2.1. *In Chapter 1, a structure of a vocabulary Σ under some informal interpretation $\mathcal{I}$ was seen as an abstraction of a state of affairs of the domain. Example 2.2.1 proposed a vocabulary Σ and an informal interpretation $\mathcal{I}$ (e.g., where O represents King Filip I of Belgium, etc.). Example 2.2.2 proposed a structure $\mathfrak{A}$ for Σ. Explain $\mathcal{I}(\mathfrak{A})$; i.e., explain what potential state of affairs is described by $\mathfrak{A}$ under $\mathcal{I}$.*

Propositional structures For vocabularies Σ consisting of propositional formulas, it makes no use to include a domain in a Σ-structure, Consequently, the definition of structure can be simplified.

Definition 2.2.3. Given a propositional vocabulary Σ, a Σ-structure is a function $\mathfrak{A} : \Sigma \to \mathbb{B}$.

It is common to denote a propositional Σ-structure as the set of true symbols. E.g., for $\Sigma = \{P/0, Q/0, R/0\}$, the structure mapping P, Q to $\mathbf{t}$ and R to $\mathbf{f}$ is denoted $\{P, Q\}$.

Pay attention for overloading. A set $\{P, Q\}$ can denote a vocabulary, but in the context of a given vocabulary Σ that contains P, Q, it denotes the Σ-structure that maps P, Q to $\mathbf{t}$ and other symbols of Σ to $\mathbf{f}$. It means that $\{P, Q\}$ denotes a different structure in the context of $\Sigma = \{P/0, Q/0, R/0\}$ than of $\Sigma = \{P/0, Q/0\}$.

Herbrand structures Given a vocabulay Σ, a Herbrand Σ-structure is a term structure: in which the domain consists of all terms over Σ. Moreover, object and function symbols have a "canonical" interpretation. E.g, the value $f^{\mathfrak{A}}$ of a function symbol $f/1$ is the function that maps any term t to the term $f(t)$.

Definition 2.2.4. Given vocabulary Σ, a *Herbrand structure (or interpretation) of* Σ is a Σ-structure such that:

- $D_{\mathfrak{A}}$ is the set of all terms over Σ (to be defined in the next section);
- For each object symbol $c/0 :\in \Sigma$, it holds that $c^{\mathfrak{A}} = c$.
- For each function symbol $f \in \Sigma$, $F^{\mathfrak{A}}$ is the function that maps terms $t_1, \ldots, t_n$ to the compound term $f(t_1, \ldots, t_n)$.

In Herbrand structures, domain elements correspond to terms, functions interpreting function symbols are trivial functions mapping tuples of argument terms to the corresponding compound terms $f(t_1, \ldots, t_n)$.

All Herbrand Σ-structures have the same domain and value of all function symbols. If Σ is given, Herbrand Σ-structures are often denoted as the set of true atomic formulas $P(t_1, \ldots, t_n)$. This notation extends the way structures of propositional symbols are sometimes denoted.

Herbrand structures are important in the theory of Logic Programming languages and databases. A database instance DB contains a set of tables, each consisting of a set of cells on the intersection of rows and columns. Each cell contains an object symbol. The closest thing in logic to a database instance DB is a Herbrand structure for vocabulary Σ_{DB} consisting of the set of object symbols S_{DB} and predicate symbols P/n for every table named P with n columns.

Example 2.2.3. The following sketches is a simple database instance in the application field of the (fictious) master informatics at the university of Lilliput. It stores the current courses, lecturers, students and grades.

Instructor

Ray	CS230
Hec	CS230
Sue	M100
Sue	M200

PassingGrade

AAA
AA
A
B
C

Enrolled

Jill	CS230
Jack	CS230
Flo	CS230
Jill	M100
Flo	M100
Jill	M200
Flo	M200

Prerequ

CS230	M100
M100	M200

Grade

Jill	CS230	A
Jack	C230	C
Flo	CS230	AA
Jill	M100	D
Flo	M100	A
Flo	M200	AAA

From a logical point of view, it is seen as a Herbrand structure $\mathfrak{A}_{DB}$. E.g., $Jill^{\mathfrak{A}_{DB}} = Jill$, and $Prereq^{\mathfrak{A}_{DB}} = \{(CS230, M100), (M100, M200)\}$.

2.2.2 Formal syntax

Below we assume an infinite supply of function and predicate symbols of every arity ≥ 0.

Definition 2.2.5. We define a *term* and *formula* **by induction**:

- an object symbol is a term;
- if $t_1, \ldots, t_n$ are terms and f is a function symbol of arity n, then $f(t_1, \ldots, t_n)$ is a term;
- a propositional symbol $P/0$ is formula, called an atom;
- if $t_1, \ldots, t_n$ are terms and P is a relation symbol of arity n, then $P(t_1, \ldots, t_n)$ is a formula, called an *atom*, and $(t_1 = t_2)$ is a formula, called an *(equality) atom*;

- if α, β are formulas then $(\neg\alpha)$, $(\alpha \wedge \beta)$, $(\alpha \vee \beta)$, $(\alpha \Rightarrow \beta)$, $(\alpha \Leftrightarrow \beta)$ are formulas;
- if x is an object symbol and α is a formula then $(\exists x\ \alpha)$ and $(\forall x\ \alpha)$ are formulas; here, x is called a (quantified) variable.

An *expression* is a term or a formula.

Terms and formulas are of a different type. Terms denote objects of the domain; formulas denote propositions about the domain. This is a big difference. Please do not mix them up.

A formula $\alpha \wedge \beta$ is called a *conjunction* and α, β are called its *conjuncts*. Likewise, $\alpha \vee \beta$ is called a *disjunction* and α, β are called its *disjuncts*. A formula $\alpha \Rightarrow \beta$ is called a *(material) implication*, α is called its *premise* or *antecedent* or *condition* and β its *conclusion* or *consequent*. A formula $\alpha \Leftrightarrow \beta$ is called an *equivalence*.

Notice that standard atoms $P(t_1, \ldots, t_n)$ are written in pre-fix notation, while equality atoms are written in in-fix notation $(s = t)$ rather than $=(s, t)$. Likewise, mathematical operators $\leq, <, \geq, >, \neq, +, \times, \div$ are all written in-fix in FO. There is no reason for this except for tradition and similarity to mathematical practice.

Wat does this inductive definition mean?

Definition 2.2.5 is an example of a *monotone inductive definition*. What exactly does an inductive definition mean?

An inductive definition defines one or more concepts (sets, relations) by describing how to construct the defined set(s). Such definitions are often presented as a collection of rules, as is the case with Definition 2.2.5. The construction starts from the empty set; at each step, one or more rule instances are applied: they are instances of rules with a condition satisfied in the current set, but not the conclusion. Application of a rule instance adds the conclusion to the set. This process goes on until no applicable rule instances are left, i.e., every rule instance with satisified condition has a satisfied conclusion. This process is called the *induction process*. The defined concepts are those obtained in the limit.

E.g., Definition 2.2.5 defines two sets simultaneously: the set of terms and the set of formulas. We show a few steps of one possible induction process:

1. $$Terms = \emptyset; Formulas = \emptyset$$
2. Initially, the only applicable rule instances are instances of *base rules*: rules without conditions. These are the rules defining that object symbols are terms and propositional symbols are formulas. Let us derive one term and one formula (using object symbol O and propositional symbol ST):
$$Terms = \{O\}; Formulas = \{ST\}$$
3. Now, the applicable rule instances are other base rule instances, but also some instances of *inductive rules*. Let us derive again one term and one formula using inductive rules:
$$Terms = \{O, \underline{S(O)}\}; Formulas = \{ST, \underline{ST \Rightarrow ST}\}$$

4. ...

This process ends when there are no more applicable rules. Infinitely many steps are needed. We obtain an infinite set of terms and formulas.

Notice that the order in which rules are applied is left unspecified. Indeed, one can show that the order does not matter: any induction process constructs the same defined concepts.

Definition 2.2.5 constructs two sets simultaneously. Such a definition is sometimes called a **definition by simultaneous or mutual induction**.

Definition 2.2.5 is called **monotone**. This means that the conditions of the rules are positive. I.e., a rule only applies due to the presence of elements in the defined relation, not due to the absence of relation. We will soon see an example of a non-monotone definition.

The induction process for Definition 2.2.5 derives the infinite set of all terms and formulas. However, notice that each of them is of finite length. I.e., there are no infinite expressions. For example the infinite string $S(S(S(\dots)))$ is not a term. The infinite string $(\forall x((x = \mathfrak{0}) \vee (x = \mathfrak{1}) \vee (x = \mathfrak{2}) \vee \dots))$ is not a formula.

BNF Special purpose formats have been developed for defining syntax of a language. For example, different sorts of grammar languages. A well-known one is the **Backus Naur Form** (BNF). Definition 2.2.5 expressed in BNF defines terms t, formulas φ and expressions e:

t	$::=$	C	object symbol
	$\mid$	$f(t, \dots, t)$	functional term
A	$::=$	$P(t, \dots, t)$	atom
	$\mid$	$(t = t)$	equality atom
φ	$::=$	A	atomic formula
	$\mid$	$(\neg\varphi)$	negation
	$\mid$	$(\varphi \wedge \varphi)$	conjunction
	$\mid$	$(\varphi \vee \varphi)$	disjunction
	$\mid$	$(\varphi \Rightarrow \varphi)$	material implication
	$\mid$	$(\varphi \Leftrightarrow \varphi)$	equivalence
	$\mid$	$(\forall x\ \varphi)$	universal quantification
	$\mid$	$(\exists x\ \varphi)$	existential quantification
e	$::=$	$t \mid \varphi$	

The meaning of this BNF is the same as Definition 2.2.5. The induction process of the definition corresponds to the way terms can be formed using the grammar.

Free and bound occurrences of symbols A symbol may occur multiple times in an expression.

Definition 2.2.6. An occurrence of a symbol σ in a subformula $(\exists\sigma\alpha)$ or $(\forall\sigma\alpha)$ f an expression e is called a *bound occurrence* of σ in e. Any occurrence of a symbol σ in an expression e that is not a bound occurrence is called a *free occurrence*.

A symbol σ is called a free symbol of e if it has at least one free occurrence in e.

Example 2.2.4. Free occurrences of symbols in the following formula are underlined twice, bound occurrences only once:

$$(\forall \underline{y}(\underline{\underline{P}}(\underline{x}, \underline{y}) \vee \exists \underline{x}(\underline{\underline{Q}}(\underline{x}, \underline{\underline{C}}, \underline{y}))))$$

Consequently, the free symbols of this formula are $P/2, x/0 :, C/0 :, Q/3$.

Notice that the symbol x occurs both bound and free. How confusing! Do not ever do this while modelling. It is asking for trouble.

Expressions over vocabulary Σ While modelling in some application domain, a formal expression e is meaningful to us in the application domain, if all free symbols of it have an informal interpretation. Suppose that we have choosen a vocabulary Σ of ontology symbols, together with an informal interpretation $\mathcal{I}$, then e is meaningful if all its free symbols belong to Σ. This motivates the following definition:

Definition 2.2.7.

- An expression e is an expression *over* Σ if the free symbols of e belong to Σ.
- A formula φ is a *sentence over* Σ if the free symbols of φ belong to Σ.
- A theory T is a *theory over* Σ if it is a set of sentences over Σ.

Notice that if e is an expression over Σ, then also over any extension of Σ.

Example 2.2.5. There exists formulas without free symbols. They are sentences over every vocabulary. An example is:

$$(\exists x(\exists y(\neg(x = y))))$$

It expresses that two different objects exist. Recall that $=$ is a logical symbol.

Notational conventions For convenience, brackets may be omitted if it is clear where to re-insert them. This is governed by the following conventions:

- Brackets at the outside of the formula may be dropped. E.g., $(\forall x(\forall y(x = y)))$ may be written as $\forall x(\forall y(x = y))$.
- For inner brackets, binding rules determine the binding strength of connectors and quantifiers. Brackets need to be inserted closer to connectives with higher precedence.

  The binding precedence of different connectives is:

  $$\neg > \wedge > \vee > \Rightarrow > \Leftrightarrow > \forall > \exists :$$

  - $\neg P \wedge Q \vee R$ is shorthand notation for $(((\neg P) \wedge Q) \vee R)$.
  - $\alpha \wedge \beta \vee \neg\alpha \wedge \neg\beta$ is shorthand notation for $((\alpha \wedge \beta) \vee ((\neg\alpha) \wedge (\neg\beta)))$
  - $\exists c P(c) \wedge Q(c, d)$ is shorthand for $(\exists c(P(c) \wedge Q(c, d)))$.

  All binary connectives are right associative:

– $P \wedge Q \wedge R$ is shorthand for $(P \wedge (Q \wedge R))$, not for $((P \wedge Q) \wedge R)$.

Example 2.2.6. The following string

$$\exists x \exists y \exists z \neg(x = y) \wedge \neg(x = z) \wedge \neg(y = z)$$

is a shorthand notation for the formula:

$$(\exists x(\exists y(\exists z(\neg(x = y) \wedge (\neg(x = z) \wedge \neg(y = z))))))$$

2.2.3 Formal semantics

We define here the value of terms and formulas in structure and the truth or satisfaction relation. First we introduce a useful notation to expand a structure with a value for a new symbol.

Definition 2.2.8. Given is a structure $\mathfrak{A}$, a symbol σ and a suitable value v for σ in $\mathfrak{A}$. We denote by $\mathfrak{A}[\sigma : v]$ the structure identical to $\mathfrak{A}$ except that $\sigma^{\mathfrak{A}[\sigma:v]} = v$.

The structure $\mathfrak{A}[\sigma : v]$ is obtained by expanding $\mathfrak{A}$ with value v for σ if $\mathfrak{A}$ does not interpret σ, or overriding the value of σ in $\mathfrak{A}$ with v if $\mathfrak{A}$ interprets σ. All other symbols retain their value.

Definition 2.2.9. Let t be a term, $\mathfrak{A}$ a structure. If $\mathfrak{A}$ interprets all free symbols of t, then the *interpretation* (also called the *value*) of t in $\mathfrak{A}$, denoted $t^{\mathfrak{A}}$, is defined by induction on the structure of t:

– If t is an object symbol, then $t^{\mathfrak{A}}$ is the domain element assigned by $\mathfrak{A}$ to t.
– If $t = f(t_1, \ldots, t_n)$ then $t^{\mathfrak{A}}$ is the domain element $f^{\mathfrak{A}}(t_1^{\mathfrak{A}}, \ldots, t_n^{\mathfrak{A}})$, that is, the domain element obtained by applying the function $f^{\mathfrak{A}}$ to the tuple of values $(t_1^{\mathfrak{A}}, \ldots, t_n^{\mathfrak{A}})$.

Otherwise, if some free symbol of t is not interpreted in $\mathfrak{A}$ then $t^{\mathfrak{A}}$ is undefined.

Let $\mathfrak{A}$ interpret all free symbols of φ. We define that $\mathfrak{A}$ *satisfies* φ (denoted $\mathfrak{A} \models \varphi$) by an inductive case analysis on the structure of φ:

– $\mathfrak{A} \models P(t_1, \ldots, t_n)$ if $(t_1^{\mathfrak{A}}, \ldots, t_n^{\mathfrak{A}}) \in P^{\mathfrak{A}}$;
– $\mathfrak{A} \models (t = s)$ if $t^{\mathfrak{A}} = s^{\mathfrak{A}}$;
– $\mathfrak{A} \models (\alpha \wedge \beta)$ if $\mathfrak{A} \models \alpha$ and $\mathfrak{A} \models \beta$;
– $\mathfrak{A} \models (\alpha \vee \beta)$ if $\mathfrak{A} \models \alpha$ or $\mathfrak{A} \models \beta$ (or both);
– $\mathfrak{A} \models (\neg\alpha)$ if $\mathfrak{A} \not\models \alpha$; (that is, $\mathfrak{A}$ does not satisfy α);
– $\mathfrak{A} \models (\alpha \Rightarrow \beta)$ if $\mathfrak{A} \not\models \alpha$ or $\mathfrak{A} \models \beta$ (or both);
– $\mathfrak{A} \models (\alpha \Leftrightarrow \beta)$ if $\mathfrak{A} \models \alpha$ and $\mathfrak{A} \models \beta$ or $\mathfrak{A} \not\models \alpha$ and $\mathfrak{A} \not\models \beta$;
– $\mathfrak{A} \models (\exists x \alpha)$ if there exists $d \in D_{\mathfrak{A}}$ such that $\mathfrak{A}[x : d] \models \alpha$;
– $\mathfrak{A} \models (\forall x \alpha)$ if for all $d \in D_{\mathfrak{A}}$, it holds that $\mathfrak{A}[x : d] \models \alpha$.

Otherwise, if $\mathfrak{A}$ does not interpret all free symbols of φ then $\mathfrak{A} \models \varphi$ and $\mathfrak{A} \not\models \varphi$ are undefined.

We define the truth function $(\cdot)^{\mathfrak{A}}$ for formulas:

– $\varphi^{\mathfrak{A}} = \mathbf{t}$ if $\mathfrak{A} \models \varphi$;
– $\varphi^{\mathfrak{A}} = \mathbf{f}$ if $\mathfrak{A} \not\models \varphi$.
– $\varphi^{\mathfrak{A}}$ is undefined if $\mathfrak{A} \models \varphi$ is undefined.

Recall from Chapter 1 that if $\mathfrak{A} \models \varphi$, we say that
- φ is *true* in $\mathfrak{A}$.
- $\mathfrak{A}$ *satisfies* φ;
- $\mathfrak{A}$ is a *model* of φ.

The same terminology holds for theories T instead of formulas φ.

What does this inductive definition mean? In this definition, the interpretation function of terms $t^{\mathfrak{A}}$ and the satisfaction relation $\mathfrak{A} \models \varphi$ are defined *by induction on the structure of t and* φ. What does this mean?

This is the same sort of induction as used when defining the factorial function:

> We define the factorial function $! : \mathbb{N} \to \mathbb{N}$ by induction on the natural numbers:
> - $0! = 1$
> - $(n+1)! = (n+1) \times n!$

The definitions of $t^{\mathfrak{A}}, \mathfrak{A} \models \varphi$ and $n!$ are three examples of another type of inductive/recursive definition: **definition by induction on an induction order** $<_i$. This defines a function (or relation) for some objects, in terms of the value of the function (or relation) in strictly smaller objects. Here strictly smaller means, strictly smaller in the induction order.

E.g., in the definition of $t^{\mathfrak{A}}$, the induction order is the subterm order: $t_1 <_i t_2$ if t_1 is a subterm on t_2. The definition defines $t^{\mathfrak{A}}$ in terms of the values ${t_i}^{\mathfrak{A}}$ of subterms of t. Likewise, in the definition of $\mathfrak{A} \models \varphi$, the induction order is the subformula order and $\mathfrak{A} \models \varphi$ is defined in terms of the satisfaction of subformulas of φ. Finally, in case of the factorial, the induction order is the standard order of the natural numbers, and $(n+1)!$ is defined in terms of the factorial of $n < n+1$.

Notice that once again, Definition 2.2.9 of $\models$ is formulated as a collection of rules. However, this time it is a non-monotone definition since it contains non-monotone rules. E.g., the $\neg$-rule "$\mathfrak{A} \models \neg\varphi$ if $\mathfrak{A} \not\models \varphi$" is non-monotone: it applies if $(\mathfrak{A}, \varphi)$ is absent from the satisfaction relation. This poses a problem for the induction process.

E.g., in the initial stage of the induction process, the relation $\models$ is initialized to be empty. Which means that, at this stage, for every structure $\mathfrak{A}$ and every formula φ, $\mathfrak{A} \not\models \varphi$ is true. Hence, the $\neg$-rule instance for arbitrary $\mathfrak{A}$ and φ is applicable; applying it would derive $\mathfrak{A} \models \neg\varphi$ for arbitrary φ. Obviously, this is wrong. We should wait to derive $\mathfrak{A} \models \neg\varphi$ until the induction process has had the chance to derive whether $\mathfrak{A} \models \varphi$. If we apply the rule to derive $\mathfrak{A} \models \neg\varphi$ too early, later rule applications might still derive $\mathfrak{A} \models \varphi$ in which case we have derived both $\mathfrak{A} \models \neg\varphi$ and $\mathfrak{A} \models \varphi$, which is a contradiction.

In definitions over an induction order, the order of rule applications is not arbitrary but must follow the induction order. In the case of $\models$, a rule instance deriving $\mathfrak{A} \models \varphi$ cannot be applied as long as there are subformulas α of φ that are derivable but have not been derived.

2.2.4 Informal semantics

When we use a formal language to express a proposition of an application domain, what proposition about the application domain have we expressed? This is a basic question in every modelling language. The informal semantics of the modelling language addresses this question.

The informal semantics of terms and formulas is determined by the informal interpretation $\mathcal{I}$ of the non-logical symbols. In the case of FO, n-place function symbols represent n-ary functions, and n-place predicate symbols represent n-ary relations in the application domain. Furthermore, also the meaning of logical symbols is known, as seen in Table 2.1. It is then possible to translate each logic formula with free symbols from Σ in a proposition $\mathcal{I}(\varphi)$ about the application domain.

A proposition is a thought. A more "material" way of expressing it is by a translation into a natural language string. The informal interpretation of an ontology symbol can expressed as a natural language pattern: a noun-phrase for a function symbol; a verb-phrase for a relation symbol. It is then possible to translate each logic formula over Σ, into a syntactically correct, meaningful natural language string that expresses $\mathcal{I}(\varphi)$.

Example 2.2.7. Extending Example 2.2.1, the informal interpretation $\mathcal{I}$ of symbols can be expressed in terms of natural language patterns:

- O: "King Filip I of Belgium"
- $S(\#_1)$: "the father of $\#_1$"
- $AgeOf(\#_1)$: "the age of $\#_1$"
- $\#_1 + \#_2$: "the sum of $\#_1$ and $\#_2$"
- ST: "it is sunny today"
- $Q(\#_1, \#_2)$: $\#_1$ is a grand parent of $\#_2$.

Now, every FO formula φ with free symbols from this list can be translated into a natural language string. E.g.:

$$\exists x(Q(x, O) \wedge AgeOf(x) < AgeOf(S(O)))$$

"*There exists x such that x is a grand parent of King Filip I of Belgium and the age of x is less than the age of the father of King Filip I of Belgium*". This string is a syntactically correct and meaningful statement expressing the proposition $\mathcal{I}(\varphi)$ that King Filip I has a grand parent that is younger than his father.

Example 2.2.8. Take the application domain of students and courses which contains the following ontology symbols, together with NL patterns expressing the informal interpretation $\mathcal{I}$:

- T: Tom;
- M: G0B23;
- $P(\#_1)$: $\#_1$ is a passing grade;
- $G(\#_1, \#_2, \#_3)$: student $\#_1$ has grade $\#_3$ for the exam of course $\#_2$.

The sentence $\exists g(G(T, M, g) \wedge P(g))$ translates into *"There exists g such that student Tom has grade g for the exam of course G0B23 and g is a passing grade"*. This string expresses the proposition $\mathcal{I}(\varphi)$ that Tom passes for course G0B23.

Exercise 2.2.2. *Try some other formulas φ in the vocabularies of the previous examples and check out $\mathcal{I}(\varphi)$. Change the informal interpretation $\mathcal{I}$ and check out how $\mathcal{I}(\varphi)$ changes.*

The informal semantics of a logical sentence under an informal interpretation is sometimes also called its *declarative or intuitive reading.*

The translation of FO in NL presents FO as as a kind of shorthand notation for a special structured subset of NL sentences. It is no poetry, but the sort of language often found in mathematical texts. As such, it inherits the *precision* of the language used in mathematical texts to express $\mathcal{I}(\varphi)$.

In general, natural language is much more complicated than FO. Natural language is a complex system, with many exceptions, overloading, special cases, special constructions, context depen-

dencies, implicit assumptions, ... It is not for nothing that NL analysis is considered one of the most difficult problems in Artificial Intelligence.

Example 2.2.9. Compare the following statements:
- In front of the door is a mercedes.
- A mercedes is expensive.

Both sentences make a quantified statement about objects belonging to the class of mercedes cars. In the first sentence, the existential quantifier is meant, but in the second, the universal quantifier: some mercedes car is in front of the door, while all mercedes cars are expensive. Apparently the word "a" sometimes stands for existential, sometimes for universal quantification. It is not easy to explain when one is meant or the other. ■

Example 2.2.10. Consider the following statement:

A human is adult if its age is more than 21.

Its syntactic structure is well preserved in the following FO sentence:

$$\forall h(Human(h) \Rightarrow (Adult(h) \Leftarrow AgeOf(h) \geq 21) \tag{2.1}$$

Despite the superficial structural correspondence, this is not a correct translation at all. The logic sentence is far weaker than the natural language statement. The natural language statement expresses a definition of the concept of adult. For definitions, we use particular natural language expressions that superficially resemble other types of expressions, but mean something much stronger. The accurate translation in FO is the sentence:

$$\forall h(Adult(h) \Leftrightarrow Human(h) \wedge AgeOf(h) \geq 21) \tag{2.2}$$

Definitions are often expressed in the form "An A is a B if (conditions)". The word "if" is notoriously overloaded. We will call this "if" the *definitional conditional*. What is meant is expressed best as $\forall x(B(x) \Leftrightarrow A(x) \wedge \varphi_{conditions})$. [2] ■

Exercise 2.2.3. *Compare the sentences in Equations 2.1 and 2.2. Do a possible world analysis. Search for structures that satisfy one but not the other. In your opinion, which sentence expresses best the original natural language statement?*

When we build a formal specification, we are not doing a mechanical translation from natural to formal language, but we are translating thoughts/propositions into formal expressions. The first step in writing a formal specification, is to understand the proposition(s) to be expressed. This may require a careful analysis of the intended meaning of an NL text. The second step is to express the propositions in the logic. With time and exercise, we can learn to recognize whether and how a proposition can be expressed in some vocabulary in some logic.

In Section 2.2.9, we return to the informal semantics of FO and the question of the existence of semantic mismatches in FO.

2.2.5 Values of connectives

In Definition 2.2.9, we use the natural language connectives to define the truth relation. E.g., the rule "$\mathfrak{A} \models \alpha \wedge \beta$ if $\mathfrak{A} \models \alpha$ and $\mathfrak{A} \models \beta$.". This can be avoided by relying on the truth table

[2] Check that many definitions in the course notes contain definitional conditionals.

of $\wedge$ defining a boolean function. Below are the truth tables of all connectives well known from boolean algebra:

$\neg$	
t	**f**
f	**t**

$\wedge$	**t**	**f**
t	**t**	**f**
f	**f**	**f**

$\vee$	**t**	**f**
t	**t**	**t**
f	**t**	**f**

$\Rightarrow$	**t**	**f**
t	**t**	**f**
f	**t**	**t**

$\Leftrightarrow$	**t**	**f**
t	**t**	**f**
f	**f**	**t**

Using these truth tables, one can recursively compute the truth value of a propositional formula in a structure $\mathfrak{A}$.

2.2.6 Derived semantical concepts

In Chapter 1 a number of derived semantics concepts were defined for all modelling logics. Let T be a theory over Σ.

- T is *satisfiable* or *semantically consistent* if there exists a Σ-structure that satisfies T.
- T is *unsatisfiable* or *semantically inconsistent* or *contradictory* if there is no Σ-structure that satisfies T.
- T is *tautologically true* or *logically valid* (notation $\models T$) if T is satisfied in every Σ-structure.
- T is *categorical* over Σ if it has a unique Σ-model (modulo isomorphism).

Let T and T' be theories or sentences. Let Σ be the vocabulary consisting of all free symbols of both.

- T is *logically equivalent* to T' (notation $T \equiv T'$) if T and T' are true in the same Σ-structures.
- T *logically entails* T' (notation $T \models T'$) if every Σ-structure $\mathfrak{A}$ that satisfies T satisfies T'.

2.2.7 Some basic properties

A simple, intuitive, vital property is that the value of an expressions e in structures $\mathfrak{A}$ only depends on the value of the free symbols of e in $\mathfrak{A}$.

Proposition 2.2.1. *Let $\mathfrak{A}$ and $\mathfrak{A}'$ be two structures with the same domain and the same value for all the free symbols of expression e. It holds that $e^{\mathfrak{A}} = e^{\mathfrak{A}'}$.*

Proof. The proof is by induction on the structure of e.

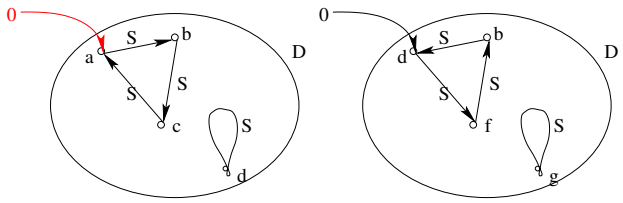
Base case: if e is an object or propositional symbol, then e is a free symbol of e and hence, the value of e in $\mathfrak{A}$ and $\mathfrak{A}'$ are identical. It follows that $e^{\mathfrak{A}} = e^{\mathfrak{A}'}$.

Inductive cases. Let e be a term $f(t_1, \ldots, t_n)$. Each free symbol of t_i is a free symbol of e and has the same value in $\mathfrak{A}$ as in $\mathfrak{A}'$. By the induction hypothesis, it follows that $t_i^{\mathfrak{A}} = t_i^{\mathfrak{A}'}$. This holds for every $i \leq n$. Since also f is a free symbol of e, its value is the same in $\mathfrak{A}$ and $\mathfrak{A}'$; hence, $f^{\mathfrak{A}}(t_1^{\mathfrak{A}}, \ldots, t_n^{\mathfrak{A}}) = f^{\mathfrak{A}'}(t_1^{\mathfrak{A}'}, \ldots, t_n^{\mathfrak{A}'})$.

The above reasoning applies also for all formulas except quantified formulas. It does not hold for quantified formulas $\exists x\ \alpha$ and $\forall x\ \alpha$ since the free symbols of this formula and of its subformula α are not the same. So, let us consider one of these cases. Take e to be $\exists x\ \alpha$. The free symbols of α are those of e plus x. Let d be an arbitrary value from the domain of $\mathfrak{A}$ or $\mathfrak{A}'$ (they have the

same domain). It holds that $\mathfrak{A}[x:d]$ and $\mathfrak{A}'[x:d]$ have the same domain and the same values for all free symbols of α. The induction hypothesis entails that $\alpha^{\mathfrak{A}[x:d]} = \alpha^{\mathfrak{A}'[x:d]}$. Since this holds for arbitrary d, it follows easily that $e^{\mathfrak{A}} = e^{\mathfrak{A}'}$. A similar reasoning applies to $\forall x\ \alpha$. ■

A second property is about structures that have the same internal structure. E.g., consider the following pair of structures:



They have the same internal structure and are abstractions of the same state of affairs. Two structures that have the same internal structure are called *isomorphic*. This concept is formally defined as follows.

Definition 2.2.10. A structure $\mathfrak{A}$ is *isomorphic* to structure $\mathfrak{A}'$ (notation $\mathfrak{A} \simeq \mathfrak{A}'$) if

- they interpret the same symbols ($\Sigma_{\mathfrak{A}} = \Sigma_{\mathfrak{A}'}$), and
- there exists a bijection $b : D_{\mathfrak{A}} \to D_{\mathfrak{A}'}$ such that
  - for each $P/n \in \Sigma$, for all $d_1, \ldots, d_n \in D_{\mathfrak{A}}$,
    $$(d_1, \ldots, d_n) \in P^{\mathfrak{A}} \text{ iff } (b(d_1), \ldots, b(d_n)) \in P^{\mathfrak{A}'}$$
  - for each $f/n :\in \Sigma$, for all $d_1, \ldots, d_n, d \in D_{\mathfrak{A}}$,
    $$f^{\mathfrak{A}}(d_1, \ldots, d_n) = d \text{ iff } f^{\mathfrak{A}'}(b(d_1), \ldots, b(d_n)) = b(d)$$

The relation "is isomorphic to", formally $\simeq$, is an equivalence relation (reflexive, symmetric, transitive). $\simeq$ partitions the class of structures in equivalence classes. Every equivalence class contains infinitely many structures. All structures in the same equivalence class are abstractions of the same states of affairs. That this is indeed the case is "shown" by the following basic theorem.

Theorem 2.2.1. *If $\mathfrak{A} \simeq \mathfrak{A}'$ then for every formula φ over $\Sigma_{\mathfrak{A}}$, it holds that $\mathfrak{A} \models \varphi$ iff $\mathfrak{A}' \models \varphi$.*

Exercise 2.2.4. *The proposition follows from the more general proposition that if b is an isomorphism from $\mathfrak{A}$ to $\mathfrak{A}'$ for every expression e over the vocabulary of these structures, it holds that $b(e^{\mathfrak{A}}) = e^{\mathfrak{A}'}$. Prove this by induction on the structure of e.*

It follows that if a theory has a model, it has infinitely many models. This forces us to redefine the notion of categorical theory: an FO theory is categorical iff all its models are isomorphic.

2.2.8 Examples

Constraints on the size of the domain The following sentences do not contain free non-logical symbols, hence they are sentences over any vocabulary including the empty one.

- *The domain contains at least one element.* The following string is not a formula ($\exists x$). Correct is:

  $$\exists x(x = x)$$

  This sentence is logically valid. Indeed, by definition of structure the domain of every structure is non-empty, and the equality relation is reflexive. Hence, in any structure $\mathfrak{A}$, there exists a domain element d and $\mathfrak{A}[x : d] \models x = x$, which entails $\mathfrak{A} \models \exists x(x = x)$.

- *The domain contains at most one object*

  $$\forall x \forall y(x = y)$$

- *The domain contains at least two objects.*

  $$\exists x \exists y \neg(x = y)$$

- *The domain contains at most two objects*:

  $$\forall x \forall y \forall z((z = x) \vee (z = y))$$

- *The domain contains exactly two objects*:

  $$\exists x \exists y(\neg(x = y) \wedge \forall z((z = x) \vee (z = y)))$$

Exercise 2.2.5. *In general, how to express that the domain contains at least, exactly, at most n elements in FO for given n?*

This shows it is possible, although inconvenient, to express for fixed n that there are exactly, at most, at least n elements in FO. However, it was shown that it is impossible to express that there exists some "variable" number of elements in FO. E.g., " *there are as many students in the auditorium A than in auditorium B*". See the section on aggregates.

The effect of "only" We express the following (very similar) statements using the predicate $drink/1$ and the object symbols $Jan/0$:, $Mieke/0$:

- Jan and Mieke attend my drink.

  $$drink(Jan) \wedge drink(Mieke)$$

- Only Jan and Mieke attend my drink.

  $$drink(Jan) \wedge drink(Mieke) \wedge \forall x(drink(x) \Rightarrow x = Jan \vee x = Mieke)$$

- At most Jan or Mieke attend my drink.

  $$\forall x(drink(x) \Rightarrow x = Jan \vee x = Mieke)$$

The words "only" and "at most" have a strong impact on the content and the structure of the formula. This is another example illustrating that translating natural language sentences to logic can be difficult.

A numerical argument to show the big difference in information content of these statements is as follows. Suppose there are hundred invitees, including *Jan* and *Mieke* for the drink. How many solutions/models are there for each sentence? For the first sentence: 2^{98}; for the second 1, for the third, 4. What a small word as "only" can do!

Group theory A *group* is a mathematical structure consisting of a domain, a (total) binary operator and neutral element that satisfies three properties: associativity, neutral element and inverse element. The vocabulary is$\Sigma = \{plus/2 :, e/0 :\}$ with intended interpretation

- $plus$: the binary group operator
- e: the neutral element

The theory is :

$$\begin{array}{rl} \forall x \forall y \forall z(plus(x, plus(y, z)) = plus(plus(x, y), z)) & //(\text{associativity})] \\ \forall x(plus(x, e) = x \wedge plus(e, x) = x) & //(\text{neutral element}) \\ \forall x \exists y(plus(x, y) = e \wedge plus(y, x) = e) & //(\text{inverse element}) \end{array}$$

A model of this theory represents a group. To see the same theory written in IDP syntax: `http://dtai.cs.kuleuven.be/krr/idp-ide/?present=group`.

Graph colouring A graph consists of vertices and directed edges between them. An undirected graph, with undirected edges, is a typically defined as a symmetric directed graph. A graph colouring of an undirected graph is a mapping from vertices to colours such that adjacent vertices have different colour.

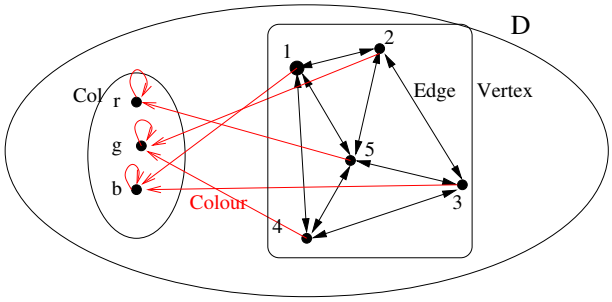
We use $\Sigma = \{Vertex/1, Edge/2, Col/1, Colouring/1 :\}$ where

- $Vertex(x)$ - x is a vertex
- $Col(x)$ - x is a colour
- $Edge(x, y)$ - there is an edge from x to y
- $Colouring(x)$ - the colouring of (vertex) x

The theory consists of two axioms:

$$\begin{array}{l} \forall x(Vertex(x) \Rightarrow Col(Colouring(x))) \\ \forall x \forall y(Edge(x, y) \Rightarrow \neg(Colouring(x) = Colouring(y))) \end{array}$$

A model of this theory represents a graph, a set of colours and a graph colouring. The following picture shows one of the models in graphical notation:



Here, a double sided arrow $i \leftrightarrow j$ represents 2 directed edges: from i to j and back.

Striking in this picture is that also colours have a colouring. It shows a representational weakness of FO. There are two natural types of objects: vertices and colours. FO is untyped, but to some extend types can be expressed with predicates $Vertex, Col$. However, function symbols in FO represent total functions on the (untyped) domain. Hence, in any structure, the function $Colouring$ is defined on colours.

How to deal with it? One way is to ignore the values of $Colouring^{\mathfrak{A}}$ outside the intended domain $Vertex^{\mathfrak{A}}$. The value of $Colouring$ in r, g, b is unconstrained and can be chosen arbitrarily. This usually does no harm but it leads to many redundant models (a factor $\times 8^3$). This problem will be avoided when extending FO with typing.

Exercise 2.2.6. *Explain the informal semantics of the following formula and verify whether it is true in the above structure.*

$$\begin{aligned}\forall x(Vertex(x) \Rightarrow \exists y \exists z(\; & Vertex(y) \wedge Vertex(z) \wedge \\ & \neg(x = y) \wedge \neg(y = z) \wedge \neg(x = z) \wedge \\ & Edge(x, y) \wedge Edge(y, z) \wedge Edge(z, x)))\end{aligned}$$

Selecting concepts and information When building a modelling of some application domain to solve one or more problems, it is important to select the concepts that are *relevant* for the problem. In the exercise below, the input stories contains concepts and information that are irrelevant for the ultimate goal.

Exercise 2.2.7. *A zoo puzzle:*

> *Mrs. Robinson's 4th grade class takes a field trip to the local zoo. The day was sunny and warm – a perfect day to spend at the zoo. The kids had a great time and the monkeys were voted the class favorite animal. The zoo had four monkeys – two males and two females. It was lunchtime for the monkeys and as the kids watched, each one ate a different fruit at its own (different) favorite resting place. Sam, who doesn't eat bananas, likes sitting on the grass. The monkey who sat on the rock ate the apple. The monkey who ate the pear didn't sit on the tree branch. Anna sat by the stream but she didn't eat the pear. Harriet didn't sit on the tree branch. Mike doesn't eat oranges.*
>
> *What did each ape eat and where? And which ape was voted favorite animal?*

Which concepts do you make explicit, which concepts are abstracted away? Which pieces of information in the story are relevant, which are irrelevant? Propose a smallest vocabulary, suitable to solve this puzzle, write the theory expressing the (relevant) information. What kind of inference is needed to solve this problem?

Exercise 2.2.8. *After having solved previous exercise, you may have a look at its partial solution at:* `http://dtai.cs.kuleuven.be/krr/idp-ide/?present=Zoo`. *This IDP specification is incomplete as can be seen from the fact that it has too many models. Extend the specification until there is only one model: the solution of the puzzle.*

Some more exercises

Exercise 2.2.9. *The first exercise shows that a simple, compact, clear NL specification, can be quite difficult to express in FO.*

> *A company as a series of potential gifts: CD, book, scarf, sunglasses, bike, company car. For this new year, it gives all its employees a CD or a book, whatever they prefer. It gives its senior employees a pair of sunglasses or a scarf. It also organizes a lottery for all employees with a bike as prize. You may assume that every employee accepts the gift of his or her choice.*

Model the situation after new year. Your theory should specify one or more axioms about a predicate symbol Receives(x, y) (employee x received gift y) such that: each model represents one possible state of affairs after all gifts have been distributed.

Try your solution at `http://dtai.cs.kuleuven.be/krr/idp-ide/?present=NewYear`. *Take care to check that nobody has less or more than deserved.*

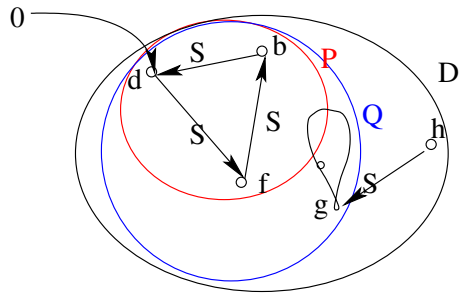
Exercise 2.2.10. *The graph G is an undirected transitive graph that contains a clique of size 3. Express this property in FO. For your information: a clique is a subset of nodes such that each node in it has edges to all other nodes in the clique. One problem here is that relations in FO are directed. So, the first question is how to encode an undirected graph? Search two solutions for this.*

Exercise 2.2.11. *Suppose that for the predicate $P/3$, the following functional dependencies exist: $1 \to 2$, and $1 \to 3$, meaning that both second and third argument are functionally dependent of the first. Express this in FO. Given that $P/3$ has these functional dependencies, this means that $P/3$ can be decomposed in two separate predicates $Q/2$ and $R/2$ that are the projections of P on respectively 1st and 2nd argument, and 1st and 3rd. Define Q and R in terms of P, and express in logic that this decomposition is equivalence preserving.*

Exercise 2.2.12. *What does the following proposition mean:*

$$P(0) \wedge \forall x(P(x) \Rightarrow P(S(x))$$

Verify whether it is true in the following structure. And if we substitute Q for P?



Express that P is a subset of Q as a formula. Try to express that g cannot be reached from 0 through the function S. Does it work?

2.2.9 Pragmatics of using FO for KR

The term *"pragmatics"* refers to the practical methodology of modelling in a logic.

When we represent knowledge in logic, the best way is to start from a clear understanding of the information we want to express. I know, a clear understanding is a thought in our brain, and so this advice does not seem to help much. The alternative is to base the methodology on how to translate natural language sentences to logic. The problem with this is that natural language is very unsystematic. We already saw a few examples of misleading NL statements at the end of Section 2.2.4 (confer "a human is adult if the age is more than 21"). Nevertheless, below I extend this list with some other NL anomalies.

Inclusive versus exclusive disjunction In NL, disjunction is overloaded. It can be intended both inclusive or exclusive. Usually, we can guess which one is meant. E.g., my brain tells me that "The test is on Monday or Tuesday" uses exclusive disjunction, while "Bill or Hillary will be home" uses inclusive disjunction.

In FO, $\vee$ is inclusive disjunction. There are multiple ways to express exclusive disjunction between ψ, φ:

$$(\psi \vee \varphi) \wedge \neg(\psi \wedge \varphi)$$
$$(\psi \wedge \neg\varphi) \vee (\neg\psi \wedge \varphi)$$
$$\psi \Leftrightarrow \neg\varphi$$

Duality: pushing negation If some proposition is not true, then what is true? It is always possible and often useful to transform the negation of a proposition into a positive proposition. This is done by pushing a negation through connectives and quantifiers in a formula. That is, a formula $\neg\varphi$ can be transformed in a logically equivalent formula φ' in which $\neg$ occurs only in front of atoms. This transformation changes some logical connectives or quantifiers to their *dual*:

- $\wedge$ to $\vee$, $\vee$ to $\wedge$,
- $\forall$ to $\exists$, $\exists$ to $\forall$,
- $\neg\neg\varphi$ to φ,
- $\varphi \Rightarrow \psi$ to $\varphi \wedge \neg\psi$,
- $\varphi \Leftrightarrow \psi$ to $\varphi \Leftrightarrow \neg\psi$.

Example 2.2.11.

$$\neg[\forall x(Vertex(x) \Rightarrow \exists y(\ Edge(x,y) \wedge \neg(x = y)))]$$

becomes

$$\exists x(Vertex(x) \wedge \forall y(\ \neg Edge(x,y) \vee (x = y)))$$

Introducing $\Rightarrow$, this is logically equivalent to:

$$\exists x(Vertex(x) \wedge \forall y(\ Edge(x,y) \Rightarrow (x = y)))$$

Exercise 2.2.13. *Take the following proposition: "For every regular language L exists a pumplength d such that for each string $s \in L$ of length at least d, there exists a partitioning of s in three parts x, y, z such that $s = xyz$ and for each $n \geq 0$, the string $xy^n z$ obtained by pumping y n times, belongs to L. ". What is true if this statement is not true? Write the negation of this sentence in a positive way. You should be able to do this, even if you do not understand the above sentence.*

Quantification Extracting quantifiers from text and expressing them in FO is sometimes challenging for several reasons.

- Quantifiers in natural language are often implicit, e.g., in words like "a". Sometimes, quantifier expressions in natural language are ambiguous.
- Quantification in FO is over the entire domain. This is called unary quantification. In contrast, quantification in natural language is virtually never over the entire universe but over some given subdomain. This is called *binary quantification* or restricted quantification. Thus, in FO we need to simulate binary quantification by unary quantification.
- The order of quantifiers in natural language text does not always correspond to the logical order.
- There are many more quantifiers in natural language than in FO.

Below these phenomena are discussed.

Overloaded quantification Natural language has implicit ways of quantifying. Recall the sentences "a mercedes is in front of the door" versus "a mercedes is expensive.". It shows that the word "a" may act as an existential quantifier as well as universal one. Context determines which one is meant.

Binary quantification simulated by unary quantification In FO, we have *unary quantification* over the entire domain as expressed in the semantic rule:

$$\mathfrak{A} \models \forall x\psi \text{ iff for all } d \in D_{\mathfrak{A}}, \ldots$$

In natural language, quantification is almost never over the entire universe as in. Instead, we use *binary quantification* in natural language: we quantify over some subclass:

All lectures of this course take place in the first semester

This sentence quantifies over the set of lectures of this course. In natural language quantification over the entire universe is virtually non-existing. Indeed, the universe is so heterogeneous that nothing sensible can be said about all entities of the entire universe.

In binary quantifications, we recognize three parts:

All **lectures of this course** take place in the first semester.

This statement has the following structure:
- a **qualification** : the group of objects we quantify over;
- a *quantor* (for all, there exists, at least three, ...);
- an assertion: the proposition stated about the quantified objects.

In FO we simulate binary quantification by unary quantification.
- All P's are Q's.

  $\forall x(\neg P(x) \vee Q(x))$ or equivalently,
  $\forall x(P(x) \Rightarrow Q(x))$

  For every object in the domain, it does not satisfy the qualification or it satisfies the assertion.

– Some P is a Q.

$$\exists x(P(x) \wedge Q(x))$$

There exists an object in the domain that satisfies the qualification as well as the assertion.

There is a striking assymmetry between expressing binary universal and existential quantification. It is a common mistake that students switch between these expressions.

Exercise 2.2.14. *Explain the difference in meaning between "All P's are Q's" and*

$$\forall x(P(x) \wedge Q(x))$$

Explain the difference in meaning between "Some P is a Q" and

$$\exists x(P(x) \Rightarrow Q(x))$$

The order of quantifiers Switching the positions of different quantifier is sometimes equivalence preserving, sometimes not. Sometimes the order of quantifiers in NL does not correspond to the logical order.

Switching different quantifiers is not equivalence preserving. E.g.:
– There is a key that fits every car. ($\exists k \forall c$)
– For every car there is a key that fits it.($\forall c \exists k$)

Exercise 2.2.15. *Give a formal proof that switching quantifiers in FO is not equivalence preserving. (Easy; one counterexample suffices.)*

However, switching equal quantifiers is equivalence preserving. E.g., $\exists x \exists y \psi$ is logically equivalent to $\exists y \exists x \psi$. Likewise $\forall x \forall y \psi$ is logically equivalent to $\forall y \forall x \psi$. This corresponds with natural language usage. The following sentences mean the same:
– There is a person living in some house. ($\exists p \exists h$)
– There is a house in which some person lives. ($\exists h \exists p$)

Exercise 2.2.16. *Prove that $\exists x \exists y \psi$ is logically equivalent to $\exists y \exists x \psi$.*

The natural language sequence of quantifiers is not always the logical one In the following NL sentence:

There will be a gift for every employee.

the logical order of quantifiers is $\forall x(Employee(x) \Rightarrow \exists y(Gift(x, y))$, which is the inverse of the order in the NL sentence.

Nested quantifiers Quantified subsentences can be nested. Consider the following sentence:

"At least one car drives faster than every car that is conducted by a police man"

It contains three nested quantifiers. Formally:

$$\exists c(car(c) \wedge \forall c1(\ (car(c1) \wedge \exists p(Police(p) \wedge Conducts(p, c1))) \Rightarrow FasterThan(c, c1)))$$

Grammatical analysis of such NL sentences is very useful to determine the nature and scope of its quantifiers.

Other quantifiers Natural language contains many more binary quantifiers.
- Each, all, every
- Some, at least one
- No, not a single
- Not all
- At least "n", at most "n", exactly "n"
- Most, Few

Exercise 2.2.17. *Propose a new syntax to model these quantifiers, extend Definition 2.2.5 and Definition 2.2.9 with rules extending syntax and semantics of FO. Do you know other examples?*

All except the last two can be expressed using $\forall, \exists$. "Most" and "Few" are vague quantifiers.

Quantifiers are studied in the domain of *generalized quantifiers*. See `http://plato.stanford.edu/entries/generalized-quantifiers/`

Expressing definitions in FO An important use of $\Leftrightarrow$ in FO is to express *definitions* of predicate and function symbols. Such formulas are called *explicit definitions*:

$$\forall x_1 \ldots \forall x_n (P(x_1, \ldots, x_n) \Leftrightarrow \psi[x_1, \ldots, x_n]$$
$$\forall x_1 \ldots \forall x_n \forall y (F(x_1, \ldots, x_n) = y \Leftrightarrow \psi[x_1, \ldots, x_n, y])$$

Here $\psi[x_1, \ldots, x_n]$ is used to denote a formula ψ that contains free variables $x_1, \ldots, x_n$.

Example 2.2.12. Express that a human is an adult if its age is at least 21:

$$\forall h (Adult(h) \Leftrightarrow Human(h) \wedge AgeOf(h) \geq 21)$$

■

Example 2.2.13. Express that a base course is a course without prerequisites:

$$\forall x (BaseCourse(x) \Leftrightarrow (Course(x) \wedge \neg \exists c : PreRequ(c, x)))$$

■

Example 2.2.14. Express that a father of a person is the male parent:

$$\forall x \forall y (Father(x) = y \Leftrightarrow Parent(x, y) \wedge Male(y))$$

■

To model a definition in FO, follow this format *closely*. Otherwise, your formula is probably wrong.

Example 2.2.15. We want to define that a person has fever if his temperature is more than 37. We write:

$$\forall x \forall t (Fever(x) \Leftrightarrow Temperature(x, t) \wedge t > 37)$$

Notice that the extra quantifier $\forall t$ does not match the format of an explicit definition, since t does not appear in the defined atom $Fever(x)$. But does it harm? It does! We analyze the formula as follows:

$\forall x \forall t(Fever(x) \Leftrightarrow Temperature(x,t) \wedge t > 37)$
$\equiv \forall x \forall t[(Fever(x) \Leftarrow Temperature(x,t) \wedge t > 37) \wedge (Fever(x) \Rightarrow Temperature(x,t) \wedge t > 37)]$
$\equiv \ \forall x \forall t(Fever(x) \Leftarrow Temperature(x,t) \wedge t > 37) \wedge$
$\quad \forall x \forall t(Fever(x) \Rightarrow Temperature(x,t) \wedge t > 37)$

The second transition is because the quantifier $\forall$ distributes over $\wedge$. That is, $\forall x(\psi \wedge \phi)$ is logically equivalent to $\forall x\psi \wedge \forall x\phi$.

Now, we focus on the second conjunct

$$\forall x \forall t(Fever(x) \Rightarrow Temperature(x,t) \wedge t > 37)$$

Because variable t does not occur in its premise, it is logically equivalent to:

$$\forall x(Fever(x) \Rightarrow \forall t(Temperature(x,t) \wedge t > 37)$$

This says that if some x has fever, every object t in the universe is the temperature of x and also, that t is also larger than 37. This is totally wrong! Correct is:

$$\forall x(Fever(x) \Leftrightarrow \exists t(Temperature(x,t) \wedge t > 37))$$

■

We will later extend FO with a language construct useful to express complex definitions.

Not all equivalences are definitional. E.g.,

$$\forall x(Human(x) \Rightarrow ((\exists y Father(x,y)) \Leftrightarrow (\exists y Mother(x,y)))$$

This statement correctly expresses that every human has a father if and only if it has a mother. Adam and Eve have neither and all other humans have both. It is not a definition.

Exercise 2.2.18. *Consider the following definition in natural language: "A human is a parent if he or she has at least one child". Consider the following formalization:*

$$\forall x(Human(x) \Rightarrow (Parent(x) \Leftrightarrow \exists c HasChild(x,c))$$

Show this is wrong by a formal possible world analysis: find a structure in which the definition and the formula disagree. Write the correct explicit definition in FO.

Exercise 2.2.19. *We define Even to be the set of all two-folds of natural numbers Nat using the following equivalence:*

$$\forall n(Even(2 \times n) \Leftrightarrow Nat(n))$$

Does this follow the format of explicit definition? If not, show where it goes wrong and write the correct explicit definition.

Exercise 2.2.20. *We intend to define narcistic love as self love. We express this in FO as:*

$$\forall x(NarcisticLove(x,x) \Leftrightarrow Love(x,x))$$

Completely wrong! Analyze it, correct it. See: `http://dtai.cs.kuleuven.be/krr/idp-ide/?present=BadEquiLove`

Exercise 2.2.21. *We define that P is the set of all sums $n+m$ with $n, m \in Q$:*

$$\forall n \forall m(P(n+m) \Leftrightarrow Q(n) \wedge Q(m))$$

Completely wrong! Analyze it, correct it. See: `http://dtai.cs.kuleuven.be/krr/idp-ide/?present=BadEquiSum`

Expressing common implicatures in FO Humans possess much shared information and exploit this during communication by not making information explicit. Propositions that are not explicitly stated but intended anyway are called *implicatures*.

Implicatures strongly depend on context and shared knowledge. Processing of implicatures is often at the subconscious level.

Predicate logic does not make hidden assumptions. In FO, all implicatures need to be made explicit and expressed. As such, modelling in logic makes us aware of our implicatures.

Example 2.2.16. Suppose that John says:" I have a son called Roger and a daughter called Ann". What information does he probably intend to provide? The names and gender of two of his children? Or the names en gender of all his children? The second one!

The proposition stated corresponds to the logic formula

$$Son(John, Roger) \wedge Daughter(John, Ann)$$

The implicature is

$$\forall x(Son(John, x) \Rightarrow x = Roger) \wedge$$
$$\forall y(Daughter(John, y) \Rightarrow y = Ann)$$

The transmitted information is the conjunction of both. The information content of what was not said (i.e., the implicature) is much larger than what he said. The implicature says that Bob, Ray, Sam, and infinitely more people are not sons. ■

Implicatures in databases Consider a table from a database:

Instructor/2	
Ray	CS230
Hec	CS230
Mar	HD87

How to express it in FO? Use $\Sigma = \{Instructor/2, Ray, Hec, Mar, CS230, HD87\}$.

The first idea that comes to mind is:

$$Instructor(Ray, CS230) \wedge Instructor(Hec, CS230) \wedge Instructor(Mar, HD87) \qquad (2.3)$$

Is this correct? Most likely not. Implicatures underlying the table are:
- different symbols $Ray, Hec, CS230, \ldots$ represent different objects;
- tuples not in the table do not belong to the $Instructor$ relation.

In what sense can we see that the database assumes such additional information? By looking at the answers that it generates for certain queries.
- $\neg Instructor(Ray, HD87)$: answer: yes
- $\exists x(Instructor(x, CS230))) \wedge x \neq Ray)$: answer: yes

A database system would answer these queries positively. For the first answer, it is only correct if the database assumes that the instructor table contains all true tuples, and that CS230 and HD87 are different objects. These are the implicatures of database tables. In FO, we have to make these implicatures explicit.

Proposition 2.2.2. *The sentence 2.3 does not entail these two propositions.*

Proof. It suffices to provide a model of sentence 2.3 in which both propositions are false.

Choose $\mathfrak{A}$ as follows:
- Domain $D_{\mathrm{U}} = \{a\}$
- $Ray^{\mathfrak{A}} = Hec^{\mathfrak{A}} = \cdots = CS230^{\mathfrak{A}} = HD^{\mathfrak{A}} = a$
- $Instructor^{\mathfrak{A}} = \{(a, a)\}$

It easy to verify that in $\mathfrak{A}$, the formula 2.3 is true. On the other hand, the two "query" formulas are false. E.g., $\mathfrak{A} \models \neg Instructor(Ray, HD87)$ iff $(Ray^{\mathfrak{A}}, HD87^{\mathfrak{A}}) \notin Instructor^{\mathfrak{A}}$ iff $(a, a) \notin \{(a, a)\}$.

■

For a correct modelling, two sorts of axiomas are needed: *UNA* and an explicit definition of *Instructor* that describes what is in and what is out the table.

UNA axiom Unique Names Axiom. This is an equality axiom to express that different object symbols denote different objects.

Let Σ be a set of object symbols. Then $UNA(\Sigma)$ is the conjunction of all disequality literals:

$$\neg(C_1 = C_2)$$

where $C_1, C_2 \in \Sigma$ are different symbols. Later we will see how to generalize it to vocabularies including functions.

Predicate completion For the predicates, we need an axiom that expresses which tuples are in it and which are out:

$$\begin{aligned}\forall x \forall y(Instructor(x, y) \Leftrightarrow\ & (x = Ray \wedge y = CS230) \vee \\ & (x = Hec \wedge y = CS230) \vee \\ & (x = Mar \wedge y = HD87))\end{aligned}$$

This is an explicit definition. We call this formula the *completed definition* or the *completion* of the table of $Instructor/2$.

The combination of UNA and the completed definiton can be shown to capture all the information in the table.

Identifiers in modelling In modelling, many symbols are chosen as *identifiers* for objects. An implicature is that different identifiers represent different objects. In pure FO, we need to make this implicature explicit by adding UNA axioms for all identiers. If there are many such symbols, this is inconvenient. In other languages such as logic programming, all object symbols are viewed as identifiers. That is, UNA is logically assumed for all object symbols.

In the IDP system, UNA is not standardly applied to a object symbol. The exception is in constructed types. E.g., when writing in the IDP

```
type day constructed from { mon; tue; wed; thu; fri; sat; sun }
myFreeDay:day
```

This logically means:
- $\forall x(day(x) \Leftrightarrow x = mon \vee \cdots \vee x = sun)$
- $UNA(\{mon, \ldots, sun\})$

On the other hands, `myFreeDay` is not an identifier. To express that my free day is Saturday or Sunday:
`myFreeDay=sat` $\vee$ `myFreeDay=sun`.

UNA is not applied to `myFreeDay`, otherwise the latter proposition would be inconsistent.

Expressing partial Functions FO assumes that a function symbol denotes a total function. In many application domains, functions are not total but partial. Recall the colouring function in the graph colouring example. Sometimes, a better representation is obtained by introducing types so that the function is total on a type but this solution is not always possible. A general solution to represent a partial function F/n :, is to use instead an $n+1$-ary predicate symbol $G_F/n+1$ called its *graph predicate*.

Example 2.2.17. Take the function that maps people to their shoesize. This is a partial function. It is only defined for people with feet. E.g., suppose we use a function and express $Shoesize(Bob) < 44$. Does this mean that Bob has feet? This is not clear. We can use a graph predicate and write $\exists x(G\_Shoesize(Bob, x) \wedge x < 44)$. This clearly implies that Bob has feet.

The conditional and material implication One of the most overloaded NL connectives is "if ... then ...". This is called the *conditional* in linguistics. In linguistic studies, more than 30 different meanings have been distinguished. Material implication, as embodied in FO, captures just one of these meanings. Unavoidably, some conditional statements in natural language will not be correctly represented by material implication in FO.

We have already seen an example: the definition conditional. The statement "A human is an adult if its age is at least 21" contains the **definitional conditional**. As we saw, it has a meaning quite different than material implication.

Below is an example of a third sort of conditional.

Compare:
- *If you succeed for all courses, then you pass.*
  - Proposed formalization:
    $$(\forall c\ Succ(c)) \Rightarrow Pass$$
- *There is a course such that, if you succeed for it, you pass*
  - Proposed formalization:
    $$\exists c(Succ(c) \Rightarrow Pass)$$

Are the NL statements equivalent? No! E.g., our university satisfies the first but not the second NL statement: there is no course such that, if you succeed for it, you pass. Even the most important course, the master thesis, is not important enough to let a student pass if he succeeds for the thesis.

Are the proposed formalizations logically equivalent? Yes! The following derivation is a correct proof:

$$\begin{aligned}
&\exists c(Succ(c) \Rightarrow Pass)\\
&\equiv \exists c(\neg Succ(c) \vee Pass)\\
&\equiv (\exists c(\neg Succ(c))) \vee Pass)\\
&\equiv (\neg \forall c\ Succ(c)) \vee Pass)\\
&\equiv (\forall c\ Succ(c)) \Rightarrow Pass
\end{aligned}$$

Every step is correct. Given that the informal NL-propositions are clearly not equivalent, it follows that at least one of these FO sentences does not correctly formalize the corresponding NL-proposition. Indeed, the problem is with $\exists c(Succ(c) \Rightarrow Pass)$.

This paradox is similar to *Smullyans drinkers paradox*. `https://en.wikipedia.org/wiki/Drinker_paradox`

Explanation All connectives of FO are *extensional*. A connective is extensional if a formula built with it and its component formulas are evaluated in the same state of affairs. Formally, a connective is extensional if its formulas and component formulas are evaluated in the same structure. E.g., we define "$\mathfrak{A} \models \psi \Rightarrow \phi$ if $\mathfrak{A} \not\models \psi$ or $\mathfrak{A} \models \phi$", hence material implication $\Rightarrow$ is extensional. We see that $\psi \Rightarrow \phi$ and its subformulas are evaluated in the same structure $\mathfrak{A}$, hence $\Rightarrow$ is extensional. The same holds for $\wedge, \vee, \neg$.

Many phrases in natural language are *intensional*: subpropositions are evaluated in *different* states of affairs than the one in which the composite proposition is evaluated. Intensional statements are common in natural language. One of them is:

There is a course such that, if you succeed for it, you pass

Indeed, what it intends to say is that there is a fixed course c, such that *in every state of affairs where the student succeeds for c*, the student obtains his degree. One can see here that the component sentence *if you succeed for it, you pass* is evaluated in many different states of affairs, probably all states of affairs that satisfy the examination regulations at the KUL. So it is not only evaluated in the actual state of affairs. Hence, it is not extensional but intensional.

Intensional propositions cannot be expressed in FO.

Intermezzo: modal logic The branch of logic that studies intensional language constructs is *modal logic*. A historically famous example of an intentional NL sentence dates from Frege.

The morning star is the evening star but it is not necessarily the case that the morning star is the evening star"

What it expresses is: (in the actual state of affairs) the morning star is the same object than the evening star but we can imagine states of affairs in which the morning star is a different object than the evening star. This is clearly an intentional proposition. [3]

[3]Context: depending on its position relative to Earth, Venus is sometimes seen in the morning and at other times, it is seen in the evening. Old greeks distinguished between the morning star and the evening star. Later they discovered that it was the same planet.

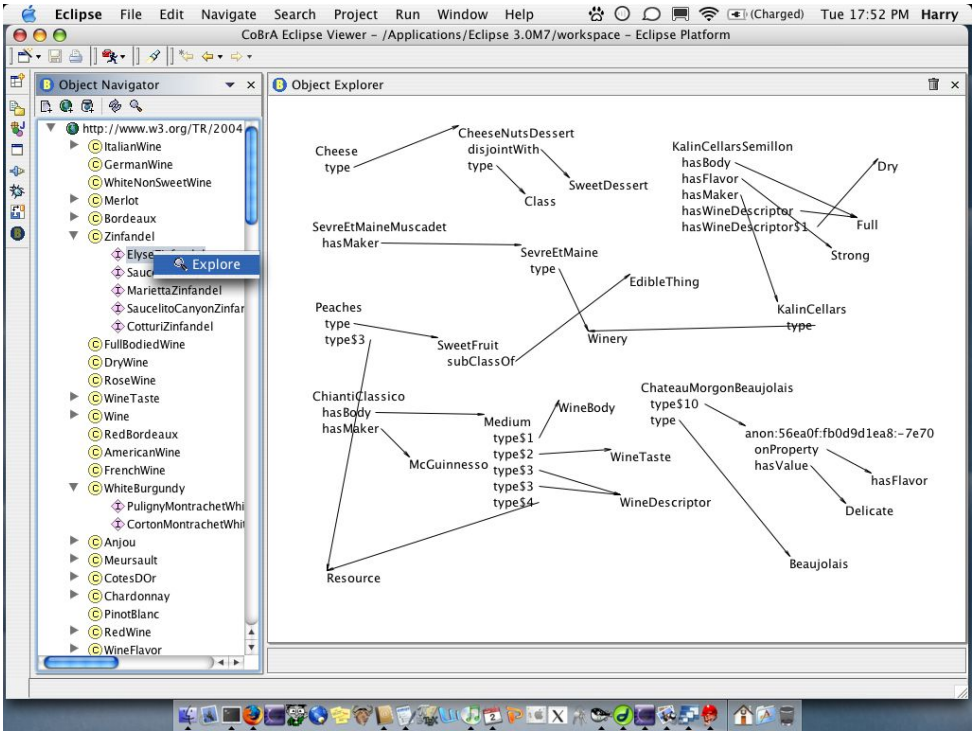


Figure 2.1: Display of the wine ontology

Why material implication? The many meanings of the conditional have caused quite some controversy about material implication. Still the material implication is probably the most commonly applied form of conditional in natural language and certainly in formal modelling. Material implication is needed for modelling.

If other conditionals are needed, then one has to add them as separate implication symbols to the logic as has been done in the context of modal logic. We will introduce another conditional as an extension of FO later in this chapter.

2.2.10 Ontologies: designing vocabularies

Designing a vocabulary involves choosing what objects and relations to express, how to model it in mathematical form, and what detail is irrelevant and can be abstracted away. Clearly, this is an important step. The result is called an *ontology*. Stated differently, an *ontology* is a set of informal concepts that are relevant to the application domain domain of discourse, plus a choice of formal non-logical symbols to represent these.

Building an ontology is important in all modelling and programming languages. E.g., in databases, ontology languages, knowledge representation languages.

Ontology languages are simple logics used to build the semantic web. They serve to express an ontology and certain simple types of logical relations between different concepts. E.g., types and subtype relationships between them, attributes of types of objects, etc. An example of an ontology language is RDF. Many tools are available for RDF, including graphical display tools. Figure 2.1 displays a graphical representation for a wine ontology in RDF available at `http://cobra.umbc.edu/images/exp_wine_ont.jpg`

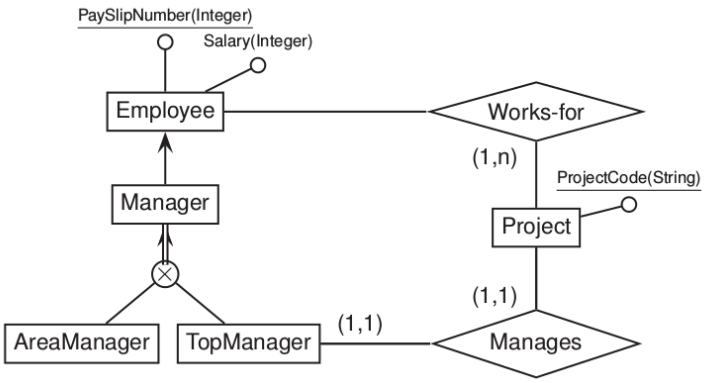


Figure 2.2: Ontology of a database: an entity-relationship schema

In the context of databases, the ontology is expressed using entity-relationship schemas, as illustrated in Figure 2.2.

Ontologies evolve When an application area evolves, the complexity of the ontology usually increases. Additional symbols are added or existing symbols are extended with new arguments. Changes to a vocabulary induce changes to the logical theory written in the vocabulary. Methodologies to minimize the impact of such changes are desirable.

Example 2.2.18. What happens with a theory containing the purchase relation $Purchase/2$, when this concept evolves over time and meta-data are added as new arguments:

- $Purchase(Bob, Volvo)$
- $Purchase(Bob, Volvo, 25000Eur)$
- $Purchase(Bob, Volvo, 25000Eur, 22/9/2017)$
- $Purchase(Bob, Volvo, 25000Eur, 22/9/2017, VolvoHeverlee)$
- ...

We observe that with each extension, each occurrence of the predicate $Purchase$ everywhere in each theory or query using this predicate is to be modified. Clearly, changing arity of a predicate has a big impact on the theory. A good methodology should avoid to do this.

Solution: *reification*: introduce an identifier for each purchase, and introduce a set of basic "attribute" relations and functions.

- $Purchase(P394)$ % P394 is identifier of the purchase
- $Buyer(P394, Bob)$
- $ItemBought(P394, Volvo)$
- $PricePaid(P394) = 25000Eur$
- ...

Adding more meta-data does not result in modifications to these predicates and function. It merely leads to more of these attribute symbols.

Example 2.2.19. An illustration of how the simple concept of marriage could evolve into a very complex one is at `https://web.archive.org/web/20170204113309/https://qntm.org/gay`.

2.3 Extensions of classical logic

FO has been extended in many ways. Below we define a number of extensions of FO, mostly those that are supported by the IDP system.

2.3.1 Extending FO with Types: FO(Types)

In standard FO, there is a unique base type where all quantification, functions and predicates range over.

Just like in other languages, *types* in FO can often be useful (1) to improve the precision of the modelling and (2) reduce the amount of human errors.

E.g., in the earlier example of graph colouring, there are two natural types: vertices and colours. The colouring function is a function from vertices to colours and is not defined on colours.

In logic, a type is sometimes called a sort. We describe multi-sorted FO, an extension of FO with a simple type system that is the basis of the type system of the IDP language.

Given a set Tb of base type symbol $\mathfrak{b}_1, \ldots, \mathfrak{b}_m, \ldots$, a type term is of the form $\mathfrak{b}_1 \times \cdots \times \mathfrak{b}_n \to \mathfrak{b}_{n+1}, n \geq 0$ where $\mathfrak{b}_i$ are base type symbols and $\mathfrak{b}_{n+1}$ may be $\mathbb{B}$, which is the boolean type.

A type expression with $\mathfrak{b}_{n+1} = \mathbb{B}$ is a *predicate type term*, any other is a *function type term*. We call such type terms also *signatures*.

A *vocabulary* Σ of multi-sorted FO consists of a set Tb of base type symbols and a set of function and predicate symbols each having a unique signature over Tb.

Structures assign domains to types and values of the right type to predicate and function symbols. Formally, a structure $\mathfrak{A}$ of vocabulary Σ consists of:
- per base type symbol $\mathfrak{b} \in \Sigma$, a domain $\mathfrak{b}^{\mathfrak{A}}$;
- per function symbol f of signature $\mathfrak{b}_1 \times \cdots \times \mathfrak{b}_n \to \mathfrak{b}$, a value $f^{\mathfrak{A}} : \mathfrak{b}_1^{\mathfrak{A}} \times \cdots \times \mathfrak{b}_n^{\mathfrak{A}} \to \mathfrak{b}^{\mathfrak{A}}$;
- per predicate symbol p of signature $\mathfrak{b}_1 \times \cdots \times \mathfrak{b}_n \to \mathbb{B}$, a value $p^{\mathfrak{A}} \subseteq \mathfrak{b}_1^{\mathfrak{A}} \times \cdots \times \mathfrak{b}_n^{\mathfrak{A}}$;

The ordered sorted logic FO(Types) is defined as the following extension of FO:

- Syntax: an expression is an FO(Types) expression if it is an FO-expression and it is *well-typed*: the type of a subexpression matches the type of the argument position in which it occurs.
- Semantics: the definition of term evaluation and satisfaction is the same as in FO, except for two modifications in the inductive rules for the quantifiers. Below, we assume that variable x has type $\mathfrak{b}_x$.
  - $\mathfrak{A} \models \forall x \varphi$ if for all $d \in \mathfrak{b}_x^{\mathfrak{A}}$, $\mathfrak{A}[x : d] \models \varphi$
  - $\mathfrak{A} \models \exists x \varphi$ if for some $d \in \mathfrak{b}_x^{\mathfrak{A}}$, $\mathfrak{A}[x : d] \models \varphi$.

A common terminology in classical logic is to call a type a *sort*. The simple form of typed logic in which sorts are disjunct, is called *many-sorted logic*. An extension is *order-sorted logic* where sorts can have subsorts.

Example 2.3.1. The IDP system supports order-sorted logic. We illustrate it in the context of graph-colouring:

```
vocabulary V{
   type Vertex
   type Col
   type BrightCol isa Col
   G(Vertex,Vertex)
   Colour(Vertex):Col
}
theory T: V{
   !  x[Vertex] y[Vertex] :  (G(x,y) => Colour(x)~=Colour(y))
   !  u v :  (G(u,v) => Colour(u)~=Colour(v))
   !  x y :  (G(x,y) => Colour(x)=y)
   }
```

Using the `isa` statement, a subtype of `Col` is defined.

The first axiom `!  x[Vertex] y[Vertex] :  (G(x,y) => Colour(x)~=Colour(y))` contains explicitly typed variables `x, y`.

In the second axiom (which is a copy of the first), no explicit types for `u, v` are given, but type inference infers that `u, v` have type `Vertex`.

For the third axiom, type inference derives a type error on the variable `y`.

FO(Types) does not add expressivity to FO. FO(types) is as expressive as FO since we can map an untyped vocabulary/structure/expression to a typed version using a single type U (the Universe).

But also the inverse is true: for the simple many and order-sorted type systems introduced above, types can be removed by making them explicit as unary predicates. This can be done in an equivalence preserving way.

E.g.,

$$\forall x[Vertex]\forall y[Vertex](G(x,y) \Rightarrow Colour(x) \neq Colour(y))$$

can be expressed as the untyped expression:

$$\forall x \forall y(Vertex(x) \wedge Vertex(y) \Rightarrow (G(x,y) \Rightarrow Colour(x) \neq Colour(y)))$$

2.3.2 FO(Types,Arit)

FO(Types) forms a basis for adding interpreted types.

FO(Types,Arit) is obtained by adding the integer numbers and operators over it as an interpreted symbols:

- type symbol *int*
- numerals: strings $0, 12, -5, \ldots$ (confer Chapter 1)
- arithmetic operators $+, \times, \hat{}$(power), $<, \leq, >, \geq, \ldots$

Their value in every structure $\mathfrak{A}$ is the obvious one.

- E.g., the value $12^{\mathfrak{A}}$ of numeral 12 in $\mathfrak{A}$ is the number 12.
- E.g., the value $<^{\mathfrak{A}}$ of symbol $<$ in $\mathfrak{A}$ is the standard strict order relation $<$ on integers, that is, $\{(n,m) \in \mathbb{Z}^2 \mid n < m\}$.

Recall the difference between a symbol and a value. Unfortunately, in text they are sometimes denoted by the same symbole. E.g., a numeral is a symbol, e.g., 5, and its value is a number, five. Likewise, the logical symbol $\leq$ denotes a value which is the mathematical relation that is denoted $\leq$, the infinite set $\{(0,0),(0,1),\ldots,(1,1),(1,2),\ldots\}$.

In FO(Types,Arit), integer arithmetic can be used in theories. E.g., Fermats last theorem is formalized as follows:

$$\neg\exists x\exists y\exists z\exists n(n > 2 \wedge x \hat{} n + y \hat{} n = z \hat{} n)$$

Also other types of numbers $\mathbb{N}, \mathbb{Q}, \mathbb{R}$ could be added to the logic in this way.

Example 2.3.2. IDP currently only supports finitely bounded arithmetic. Every numerical symbol has to be declared to be an element of a finite subtype of integers. An example is as follows.

```
vocabulary V{
   type someIntegers isa int
   C1 :  someIntegers
   C2 :  someIntegers
}
theory T: V{
   C1 + C1 = C2.
}
structure S:V{
   someIntegers = {1..5}
   }
```

2.3.3 Second and higher order logic

In FO, quantification is over elements of the domain. Second order logic (SO) is an extension of FO in which quantification over predicates (of some signature $\mathfrak{b}_1 \times \cdots \times \mathfrak{b}_n \to \mathbb{B}$) and functions (of some signature $\mathfrak{b}_1 \times \cdots \times \mathfrak{b}_n \to \mathfrak{b}$) is allowed.

The extension of syntax and semantics of FO to SO is straightforward. For syntax, we add a new rule to the definition of expression:

Definition 2.3.1.
- If φ is a formula, and P a function or predicate symbol, then $\forall P\varphi$ and $\exists P\varphi$ are formulas.

And for semantics, we extend Definition 2.2.9:

Definition 2.3.2.
- $\mathfrak{A} \models \exists P\varphi$ if P is a predicate symbol with signature $\mathfrak{b}_1 \times \cdots \times \mathfrak{b}_n \to \mathbb{B}$ and for some $X \subseteq \mathfrak{b}_1^{\mathfrak{A}} \times \cdots \times \mathfrak{b}_n^{\mathfrak{A}}$, it holds that $\mathfrak{A}[P:X] \models \varphi$.
- Likewise for function variables and universal quantifier.

Example 2.3.3. We express that $T : U \times U \to \mathbb{B}$ is the least symmetric binary relation that extends $G : U \times U \to \mathbb{B}$.

Consider the formula $\varphi[P]$ with predicate variable $P : U \times U \to \mathbb{B}$:

$$\forall x \forall y(G(x, y) \Rightarrow P(x, y)) \wedge \forall x \forall y(P(x, y) \Rightarrow P(y, x))$$

This expresses that predicate variable P extends G and is symmetric.

To express that T is the least relation that satisfies φ, two propositions are needed:

- T satisfies $\varphi[T]$:
$$\varphi[T]$$
- T is subset of every relation P that satisfies $\varphi[P]$.
$$\forall P(\varphi[P] \Rightarrow \forall x \forall y(T(x, y) \Rightarrow P(x, y)))$$
This contains a second order quantification of P.

The full formula is then:

$$\begin{array}{l} \forall x \forall y(G(x, y) \Rightarrow T(x, y)) \wedge \forall x \forall y(T(x, y) \Rightarrow T(y, x)) \wedge \\ \forall P[(\forall x \forall y(G(x, y) \Rightarrow P(x, y)) \wedge \forall x \forall y(P(x, y) \Rightarrow P(y, x))) \\ \qquad \Rightarrow \forall x \forall y(T(x, y) \Rightarrow P(x, y))] \end{array}$$

In FO and SO, predicates and functions range over elements of some domain of atomic objects. In higher order logic (HO) predicates and functions may range over sets and functions on the domain, and on sets or functions of them, etc. We will not define this logic here.

The expressivity of SO is (much) higher than of FO. That of HO is (much) higher than of SO. Later in this course, we will see several properties that can be expressed in SO but not in FO.

Exercise 2.3.1. *Given $\Sigma = \{0 : U, S : (U \to U)\}$ of a object symbol 0 and a unary function S express that the domain is a subset of very set that contains 0 and is closed under application of S.*

Exercise 2.3.2. *Express in SO: there is a group of boys and girls that went dancing, and each of the boys danced with each of the girls.*

2.3.4 Extending FO with aggregates

Sometimes, it is useful to be able to talk about properties of a set, e.g., its cardinality or its maximal element. Such attributes are called *aggregates*.

E.g., assume we want to express the proposition that the room in which a course is taught must be larger than the number of students enrolled in the course. This sentence cannot (naturally) be expressed in FO because it refers to the cardinality of a set, the set of students registered for the course. A useful extension here is one with set expressions and cardinality operator:

$$\forall c \forall r(Course(c) \wedge Room(c, r) \Rightarrow \#\{y : Enr(y, c)\} \leq Capac(r))$$

Here # is the cardinality aggregate symbol and represents the (partial) function from sets to the number of elements of the set. The symbol takes as argument a set expression $\{(x_1, \ldots, x_n) : \varphi\}$.

Useful aggregate expressions are:

- $\#\{(x_1, \ldots, x_n) : \varphi\}$: represents the number of elements.
- $Min\{x : \varphi\}$ represents the least number of a set of integers.
- $Max\{x : \varphi\}$ represents the largest element of a set of integers.
- $Sum\{(x, x_1, \ldots, x_n) : \varphi\}$ represents the sum of the first arguments of these tuples:

$$Sum\{(x, x_1, \ldots, x_n) : \varphi\} = \sum_{\{(x,x_1,\ldots,x_n):\varphi\}} x$$

Example 2.3.4. Expressing that the study load of a student should be less than 65 study points using the sum aggregate:

$$\forall s(Sum\{(l, c) : SelCourse(s, c) \wedge StudyPoints(c, l)\} \leq 65)$$

In words: the sum of the first arguments of all tuples in the set of tuples (l, c) such that l is the number of study points of a course followed by a student s should be less than 65, for each student s.

Example 2.3.5. Defining the total studyload $StudyL/1$: of a student:

$$\forall s \; StudyL(s) = Sum\left\{(l, c) : \begin{pmatrix} SelCourse(s, c) \wedge \\ StudyPoints(c, l) \end{pmatrix}\right\}$$

If we sum only over the first arguments of tuples, why do we need the other arguments?

Example 2.3.6. Consider the following expressions:

$$\forall s(Sum\{(l, c) : SelCourse(s, c) \wedge StudyPoints(c, l)\} \leq 65)$$

$$\forall s(Sum\{l : \exists c(SelCourse(s, c) \wedge StudyPoints(c, l))\} \leq 65)$$

It is simpler but is it equivalent?

No. Imagine that a student *Bob* follows 10 courses c1,..,c10 of 8 study points each. His total load is 80, far above the threshold.

Now compare the values of the sets $\{(l, c) : SelCourse(Bob, c) \wedge StudyPoints(c, l)\}$ and $\{l : \exists c(SelCourse(Bob, c) \wedge StudyPoints(c, l))\}$ in this situation. What is the sum of these sets? Respectively 80 and 8. Therefore, the second constraint is wrong. By adding more arguments, we can distinguish between different occurrences of the same study load for different courses.

Extending FO to FO(Agg) We extend the inductive Definition 2.2.5 of expression with new inductive rules and a new sort of expression.

Definition 2.3.3 (of expression)**.** – ...
- If $\bar{x}$ is a tuple of variable symbols, φ a formula, then $\{\bar{x} : \varphi\}$ is a *set expression* (also called a *set comprehension*).
- If s is a set expression and Agg is an aggregate symbol (e.g., $\#, Sum, Min, \ldots$) then $Agg(s)$ is a term (called an aggregate term).

To extend the evaluation function and satisfaction relation $\models$, we assume that each aggregate function Agg has a value $Agg^{\mathfrak{A}}$ in structure $\mathfrak{A}$. E.g., $\#^{\mathfrak{A}}$ is the function that maps sets of (tuples of) domain elements to the number of its elements. Then we add two inductive rules to Definition 2.2.9:

Definition 2.3.4 (of value of term and of satisfaction relation)**.** – ...
- $\{\bar{x} : \varphi\}^{\mathfrak{A}} = \{\bar{d} \in D^n_{\mathfrak{A}} | \mathfrak{A}[\bar{x} : \bar{d}] \models \varphi\}$
- $Agg(s)^{\mathfrak{A}} = Agg^{\mathfrak{A}}(s^{\mathfrak{A}})$

(That is all it takes.)

Thus, the value of a set expression is the set of tuples of domain elements.

We again obtain a well-defined syntax, satisfaction relation, informal semantics and hence, a well-defined modelling logic.

Remark 2.3.1. In IDP, aggregates are represented in a slightly different syntax:
- number of elements of P

```
#{x,y:  P(x,y)}.
```
- sum of x+y, for all (x,y)$\in$ P

```
sum{x,y:  P(x,y) :  x+y}.
```

This corresponds to the expression $Sum\{(z, x, y) : P(x, y) \wedge z = x + y\}$.
- minimum of set { x : Q(x)& R(x)}

```
min{x:  Q(x)& R(x) :  x}.
```
- maximum :

```
max{x:  Q(x)& R(x) :  x}.
```
- Nesting is allowed, as in:

```
Pnest= sum{x[num]:x= #{y:Q(x,y)} :  x }.
```

Exercise 2.3.3. *http://dtai.cs.kuleuven.be/krr/idp-ide/?present=Agg*
Experiment with different input/output.
- *Compute value aggregate expressions from values for P, R.*
- *Compute values of P, R from value aggregates expressions.*
- *Compute minimal structure satisfying aggregates expressions.*

2.3.5 Adding definitions to FO

Definitions are a basic component of scientific and mathematical knowledge. They are an important form of human knowledge in virtually all domains of human expertise. While building a formal specification, it is frequently useful to introduce a new concept and define it in terms of existing ones.

In FO, non-recursive definitions of predicates and functions can be expressed using explicit definitions. However, in general inductive definitions cannot be expressed. Here we present an extension of FO with an expressive definition construct.

What is a definition, informally? A *definition* specifies a defined concept in terms of other concepts, which I will call the *parameter concepts* of the definition. If the values of the parameter

concepts are fixed, then the definition determines the value of the defined concept. As such, a definition determines a *function* from values of the parameter concept(s) to values of the defined concept. If the set of parameter concepts is empty or their value is fixed, a definition specifies a unique value for the defined concept.

The defined concept is sometimes called the *definiendum*, the expression that defines it is called the *definiens*.

We can view a definition as a proposition of a special kind. It posits a "functional" kind of logical relationship between the defined concept and the parameter concepts. Functional in the sense that, for each chosen assignment of values to the parameter, a value for the defined concept exists that satisfies the definition and moreover, this value is unique.

Here, we will extend FO with a language construct to express definitions. It is rule based, supports the most common forms of inductive definitions found in mathematical text, and is supported by the IDP system. The extension of FO that we introduce below is called FO(ID).

Example 2.3.7. "*x is a brother of y if x is male, x and y differ and x and y have the same parents*"

- The *definiendum*: the brother relation
- Parameter concepts: male and parent relationships
- The *definiens*: x is a male, x and y differ and x and y have the same parents

This definition induces for any value for the male and parent relationship a unique value for the brother relationship.

Expressing definitions in FO As mentioned before, some definitions can be expressed in FO through *explicit definitions* of predicates and functions:

$$\forall x_1 \dots \forall x_n (P(x_1, \dots, x_n) \Leftrightarrow \psi[x_1, \dots, x_n]$$
$$\forall x_1 \dots \forall x_n (F(x_1, \dots, x_n) = y \Leftrightarrow \psi[x_1, \dots, x_n, y])$$

Expressing definitions in FO has two disadvantages: it is does not work in general for inductive definitions and it does not preserve the modular structure of many definitions.

The modular structure of definitions Definitions frequently consists of cases. An example is a definition of a set or relation *by exhaustive enumeration*. This is a definition that sums up the elements of the set, case by case. E.g., a weekday is Monday or Tuesday or ... or Friday. E.g., consider the following tabel:

Instructor/2	
Ray	CS230
Hec	CS230
Mar	HD87

It defines the Instructor relation by exhaustive enumeration. Every row is a case in the definition. Definitions by exhaustive enumeration have no parameters, hence, they define a unique value.

The modular structure of definitions is also prominent in inductive and recursive definitions, as can be seen in several fundamental definitions earlier in this course such as Definition 2.2.5 of

term and formula and Definition 2.2.9 of the truth relation. Inductive definitions are formulated as a set of base cases and inductive cases.

To preserve the modular structure of definitions, this suggests to express a definition as a set of rules.

A formal definition construct

Definition 2.3.5. A definition Δ is a *set of definitional rules*:

$$\forall \bar{x}(P(\bar{t}) \leftarrow \varphi)$$

where
- $\bar{x}$ is a sequence of variables $x_1 \dots x_m$, universally quantified,
- $\bar{t}$ is a tuple of n terms, with n the arity of P,
- φ is a FO-formula,
- both $\bar{t}$ and φ have free variables amongst $\bar{x}$.

Notice that the definition construct is only defined for predicates.

A predicate P in the head of a rule is called a *defined predicate* of Δ. Any other non-logical symbol that appears in a rule body is called a *parameter* or an *open symbol* of Δ. The symbol $\leftarrow$ is called the *definitional implication*. It is a conditional but one with a different meaning than the material implication! A definition is called *inductive* or *recursive* if one of its defined predicates occurs in the body of a rule.

Before we discuss the semantics, let us first review some examples to show the way informal definitions are formalized as FO(ID) definitions.

Example 2.3.8. In FO(ID) the table for the Instructor relation is expressed as follows:

$$\left\{ \begin{array}{l} Instructor(Ray, CS230) \leftarrow \\ Instructor(Hec, CS230) \leftarrow \\ Instructor(Wal, HD87) \leftarrow \\ Instructor(Mar, HD88) \leftarrow \end{array} \right\}$$

Here the empty body stands for "true".

Notice that this definition looks like a set of atoms, but has a much stronger meaning. It not only states that these atoms are true, but also that (x, y) is not in the defined relation unless (x, y) matches one of the mentioned tuples.

Example 2.3.9. A non-inductive definition with 50 or so cases:

$$\left\{ \begin{array}{l} \forall x(European(x) \leftarrow Albanian(x)) \\ \forall x(European(x) \leftarrow Armenian(x)) \\ \dots \\ \forall x(European(x) \leftarrow Turkish(x)) \\ \forall x(European(x) \leftarrow Ukrainian(x)) \end{array} \right\}$$

A similar remark holds here. Although it looks like a set of implications, the meaning is much stronger: it also expresses that nobody is an European unless he or she is covered by one of the cases.

Example 2.3.10. A non-inductive definition of the concept of an uncle, with two cases: brothers of parents, and husbands of aunts.

$$\left\{ \begin{array}{l} \forall x \forall u(Uncle(x,u) \leftarrow \exists y(Parent(x,y) \wedge Brother(y,u))) \\ \forall x \forall u(Uncle(x,u) \leftarrow \exists y \exists s(\ Parent(x,y) \wedge Sister(y,s) \wedge \\ \qquad\qquad\qquad\qquad\qquad\quad Married(s,u)))) \end{array} \right\}$$

So far, we met two types of inductive/recursive definitions: monotone inductive definitions and recursive definitions, definitions by induction over some induction order. An example of the first kind is Definition 2.2.5 of term and formula.

Example 2.3.11. Another prototypical monotone inductive definition is the following:

Definition 2.3.6. The reachability set R of root A in graph G is defined inductively:
- $A \in R$;
- if vertex $x \in R$ and there is an edge from x to y in G then $y \in R$.

For expressing this informal definition in FO(ID) we use $\Sigma = \{A/0 :, G/2, R/1\}$ with the obvious intended interpretations. The formal definition is then:

$$\left\{ \begin{array}{l} R(A) \leftarrow \\ \forall x(R(x) \leftarrow \exists y(R(y) \wedge G(y,x))) \end{array} \right\}$$

It defines R in terms of parameters G, A and consist of one base rule and one inductive rule.

A monotone inductive definition is one that consists of *monotone rules*. Informally, a monotone rule adds new elements given the *presence* of other elements in the rule. On the level of the formal syntax, it means that the occurrence of the defined predicates in the body of rules is *positive*, i.e., not in the scope of negation $\neg$. A definition of this type specifies that the defined set is obtained by a process of iterated rule application, starting from the empty set.

We also saw an example of a *nonmonotone* definition: the Definition 2.2.9 of truth which is a definition by induction over the subformula induction order. It contains several inductive rules one of which is nonmonotone:

$$\mathfrak{A} \models \neg\alpha \text{ if } \mathfrak{A} \not\models \alpha \text{ (i.e., not } \mathfrak{A} \models \alpha)$$

Informally, a nonmonotone rule adds new elements given the *absence* of other elements in the rule. Here in this case, the absence of $(\mathfrak{A}, \alpha)$ in the satisfaction relation leads to adding $(\mathfrak{A}, \neg\alpha)$. The defined set is obtained by a process of iterated rule application, starting from the empty set. However, during this process, the induction order must be respected, that is, rules deriving smaller elements in the induction order must be applied before rules deriving larger elements.

A simplified example follows.

Example 2.3.12. Consider the following (uncommon) way to define the set of even numbers.

Definition 2.3.7. We define even numbers by induction on the standard order of numbers:

- 0 is even.
- $n+1$ is even if number n is **not** even.

The inductive rule is nonmonotone. The defined set is obtained by iterated rule application, where rules deriving evenness of smaller numbers must be applied before evenness of larger numbers. Under this condition, the only possible induction process is the one that derives evenness of $0, 2, 4, \ldots$.

It is formalized as follows:

$$\left\{ \begin{array}{l} \forall x(Even(x) \leftarrow x = 0) \\ \forall x(Even(S(x)) \leftarrow \neg Even(x)) \end{array} \right\}$$

Notice that the induction order is not made explicit. This will be discussed later.

So far, our example definitions defined a single concept. Some (inductive) definitions define multiple concepts simultaneously. Verify that we have seen already a mathematical definition in this course showing simultaneous induction: Definition 2.2.5 of term and formula. Recall that formulas are defined in term of terms, and we defined aggregate terms (e.g., $\#\{x : \varphi\}$) in terms of formulas.

Example 2.3.13.

Definition 2.3.8. We define even and odd numbers by simulataneous induction:
- 0 is even;
- if n is even then n+1 is odd;
- if n is odd then n+1 is even.

This is a monotone definition. The sets of even and odd numbers are both constructed by iterated rule application.

Using an appropriate vocabulary, it is formally expressed:

$$\left\{ \begin{array}{l} \forall x(Even(x) \leftarrow x = 0)) \\ \forall x(Odd(S(x)) \leftarrow Even(x)) \\ \forall x(Even(S(x)) \leftarrow Odd(x)) \end{array} \right\}$$

Here $S/1$ represents the successor function, mapping a natural number n to n+1.

Informal and formal semantics As argued in earlier discussions of informal (rule-based) definitions that appear in the course (Remarks 2.2.2 and 2.2.3), informal definitions specify that the definiens is obtained by iterated rule application.

Although this explanation focusses on inductive/recursive definitions, the principle of iterated rule application also works for non-inductive definitions, in a trivial way.

Notice that the sequence of sets produced during the induction process is monotonically growing. That is, once an element is present in a set, it remains present. Monotone rules add new elements to the defined set given the *presence* of other elements. Therefore, once a monotone rule is applicable at some stage during the induction process, the rule remains applicable during the rest of the process. Therefore, monotone rules can be safely applied as soon as they become applicable.

For monotone definitions, one can prove that the defined set is the *least* set satisfying the rules of the definition. One can also prove that the order of rule applications does not matter: every induction process constructs the same value for the defined relation.

Example 2.3.14. The induction process:

$$\left\{ \begin{array}{l} R(A) \leftarrow \\ \forall x(R(x) \leftarrow \exists y(R(y) \wedge G(y,x))) \end{array} \right\}$$

Let $G^{\mathfrak{A}}$ be the graph $\{(a,b),(b,c),(c,b),(a,e),(d,d)\}$ and $A^{\mathfrak{A}} = a$.

One induction process is as follows:
- $R := \emptyset$
- $R := R \cup \{a\}$ (base rule)
- $R := R \cup \{b\}$ (inductive rule)
- $R := R \cup \{c\}$ (inductive rule)
- $R := R \cup \{e\}$ (inductive rule)
- Saturation: every rule is satisfied.

We obtain $R = \{a,b,c,e\}$. This is the set of nodes that can be reached from $A^{\mathfrak{A}} = a$ with a finite path. The only vertex that cannot be reached is d.

Exercise 2.3.4. *Verify that other induction processes (by applying the rules in a different order) determine the same outcome.*

It is clear that for this definition, each chosen value for G and A determines a unique value for R, and the induction process constructs this. This works for finite or infinite sets G.

The situation is more complex for nonmonotone definitions with rules adding elements to the defined set given the *absence* of other elements. These rules might be applicable at early stage during the induction process when certain elements are still absent but not longer after rule applications add these elements. E.g., at the start of the induction process for the satisfaction relation $\models$, it is set to the empty set. Hence, in this initial stage, it holds that $\mathfrak{A} \not\models \varphi$ for every φ, and hence, the $\neg$-rule is applicable for every φ to derive $\mathfrak{A} \models \neg\varphi$. But this is unsafe. Later rule applications might derive $\mathfrak{A} \models \varphi$, and then it would appear that $\mathfrak{A} \models \neg\varphi$ should not have been derived. The role of the induction order is exactly to prevent such untimely rule applications of nonmonotone rules. We need to wait with applying the non-monotone rule till the induction process is finished with the subformulas. Due to nonmonotone rules, induction processes that do not respect the induction order go wrong. This was explained in page 38.

Example 2.3.15. Reconsider the definition of Example 2.3.12.

$$\left\{ \begin{array}{l} \forall x(Even(x) \leftarrow x = 0) \\ \forall x(Even(S(x)) \leftarrow \neg Even(x)) \end{array} \right\}$$

In the context of the structure $\mathbb{N}$ of natural numbers, the initial value of $Even$ (abbreviated E) is $\emptyset$. At this stage, each instance of each rule is applicable and each natural number is *derivable*. The only induction process that respects the induction order is as follows:

- $E := \emptyset$; at this stage all natural numbers are derivable, 0 is the least.
- $E := \{0\}$; at this stage, the set of derivable elements is $\{2, 3, 4, \dots\}$, the least is 2.
- $E := \{0, 2\}$;
- $E := \{0, 2, 4\}$;
- ...

The limit of this process is the set of even numbers.

If we do not respect the induction order we obtain rubbish.

- $E := \emptyset$; assume that we unsafely apply the inductive rule to derive 1, rather than applying the base rule deriving 0.
- $E := \{1\}$; from now on we apply the rules along the induction order: first 0;
- $E := \{1, 0\}$; the next derivable element is 3 (2 is not derivable), then 5, etc. The limit is the set of odd numbers augmented with 0.

The limit is $\{1, 0, 3, 5, 7, \dots\}$. All rules are satisfied in this set. Once 0 was derived, the rule deriving 1 does not apply anymore; it was applied unsafely. The induction process went wrong.

We mentioned that the defined set of a monotone definition is the least set satisfying the rules. This property is not satisfied for nonmonotone definitions. E.g., the set $\{1, 0, 3, 5, 7, \dots\}$ satisfies all rules of the above definition. Yet, the set defined by this definition, namely the set of even numbers, is not a subset of $\{1, 0, 3, 5, 7, \dots\}$.

The key idea with nonmonotone definitions is that rules should be applied only if it is safe to apply them, that is, if later derivations cannot violate the antecedent of a rule that was already applied. One can prove that every induction process that safely applies rules will construct the same defined set.

A rule based definition expresses that the value of the defined concept is the result of iterated *safe* rule applications. The normal way to ensure safe rule application is that the induction process should respect the induction order. However, there are other logical ways to enforce safe rule application based on three-valued logic.

One such a way is the *well-founded semantics* of VanGelder, Ross and Schlipf. For FO(ID) we use an extension of this semantics. We will not introduce this formal semantics here. What is important is that if a structure $\mathfrak{A}$ is a well-founded model of a definition Δ, then the values $P^{\mathfrak{A}}$ of the defined symbols can be constructed by iterated safe rule application. It follows that the well-founded semantics correctly formalizes the informal semantics of rule sets Δ as inductive/recursive definitions.

The advantage of the well-founded semantics is that there is no need to make the induction order explicit. No induction order is needed to guide the induction process for definitions with non-monotone rules.

Definition 2.3.9. We define that a structure $\mathfrak{A}$ satisfies a definition Δ (notation $\mathfrak{A} \models \Delta$) if $\mathfrak{A}$ is the *well-founded model* of Δ in the context of the values of the parameter symbols specified by $\mathfrak{A}$.

This means that the value of the defined symbols of Δ in $\mathfrak{A}$ are obtained by an induction process of safe rule applications executed from the values of the parameter symbols of Δ in $\mathfrak{A}$.

Terminology 2.3.1. Often definitions over an induction order are called *recusive definitions*.

The logic FO(ID)

Definition 2.3.10. An FO(ID) theory T is a set of FO sentences and definitions. A structure $\mathfrak{A}$ satisfies T (notation $\mathfrak{A} \models T$) if $\mathfrak{A} \models \varphi$, for every FO sentence $\varphi \in T$ and $\mathfrak{A}$ satisfies Δ, for every definition $\Delta \in T$.

Remark 2.3.2. A definitional rule *only* appears inside a definition in an FO(ID) theory, *never* as a stand-alone formula in an FO(ID) theory. A definitional rule only has meaning in the context of a set of rules that together describe the induction process. E.g., the following set of expressions is not an FO(ID) theory.

$$\forall x \forall y(G(x,y) \Rightarrow F(x) = F(y))$$
$$\forall x \forall y \forall z(G(x,y) \leftarrow G(x,z), G(y,z))$$

But the following is an FO(ID) theory:

$$\forall x \forall y(G(x,y) \Rightarrow F(x) = F(y))$$
$$\{\ \forall x \forall y \forall z(G(x,y) \leftarrow G(x,z), G(y,z))\ \}$$

Remark 2.3.3. Recall that there are many conditionals in NL. The definitional implication is formalizing a sort of conditional that is quite different than the conditional formalized by material implication (see below).

Inductive definitions with aggregates in FO(ID,Agg) The inductive definition construct can be further extended for aggregates. Consider the following inductive definition:

A company A controls company B if the total sum of the shares in company B owned by A or by companies controlled by A is more than 50%.

We use the following vocabulary:
- $Cont(x,y)$: company x controls company y.
- $OwnsSh(x,y,s)$: company x owns s shares in company y.

$$\left\{ \begin{array}{ll} \forall a \forall b(Cont(a,b) \leftarrow Sum\{(s,c) : & (c = a \vee Cont(a,c)) \wedge \\ & OwnsSh(c,b,s)\} > 0.50) \end{array} \right\}$$

Alternative representations for definitions Below, we investigate alternative representations of definitions:
- as material implications ;
- as equivalences in FO;
- as second order formulas in SO.

A) Definitions versus material implications Compare the definition :

$$\left\{ \begin{array}{l} \forall x(Parent(x,y) \leftarrow Father(x,y)) \\ \forall x(Parent(x,y) \leftarrow Mother(x,y)) \end{array} \right\}$$

with the conjunction of material implications:

$$\begin{array}{l} \forall x(Parent(x,y) \Leftarrow Father(x,y)) \wedge \\ \forall x(Parent(x,y) \Leftarrow Mother(x,y)) \end{array}$$

They have a very different meaning.

- A *material implication* is true if its premise is false or its conclusion is true. If its conclusion is true, the truth of its condition does not matter. If its condition is false, the truth of the conclusion does not matter.
- Rules in a *definition* is a case. For some defined atom to be true, it should be derived by at least one rule with a true body.

If for a set of material implications, all the premises are false, then all the conclusions are **arbitrary**, they can be true or false. In the corresponding set of definitional rules, then all the conclusions should be **false**.

That is certainly an important difference.

Exercise 2.3.5. *Here is a quantitative comparison. Assume a structure $\mathfrak{A}$ with n domain elements and empty $Father^{\mathfrak{A}}$ and $Mother^{\mathfrak{A}}$. How many possible values for Parent exist that satisfy the definition? And how many that satisfy the material implications?*

B) Predicate completion for definitions The *predicate completion* transforms a definition Δ in FO(ID) to an FO theory $Comp(\Delta)$ in the following three steps:

- Making rule heads in definitions uniform:

$$\begin{array}{c} \forall \bar{x}(P(\bar{t}) \leftarrow \varphi) \\ \downarrow \\ \forall \bar{y}(P(\bar{y}) \leftarrow \exists \bar{x}(\bar{y} = \bar{t} \wedge \varphi) \end{array}$$

  Here $\bar{y}$ is a tuple $(y_1, \ldots, y_n)$ of new fresh variables not occurring in definition. The equation $\bar{y} = \bar{t}$ stands for the conjunction $y_1 = t_1 \wedge \ldots \wedge y_n = t_n$.

- Combine sets of uniform rules for each defined P in one rule.

$$\begin{array}{c} \forall \bar{y}(P(\bar{y}) \leftarrow \varphi_1) \\ \cdots \\ \forall \bar{y}(P(\bar{y}) \leftarrow \varphi_n) \\ \downarrow \\ \forall \bar{y}(P(\bar{y}) \leftarrow \varphi_1 \vee \cdots \vee \varphi_n) \end{array}$$

- Translate each rule into a FO equivalence:

$$\begin{array}{c} \forall \bar{y}(P(\bar{y}) \leftarrow \varphi) \\ \downarrow \\ \forall \bar{y}(P(\bar{y}) \Leftrightarrow \varphi) \end{array}$$

Example 2.3.16.

$$\begin{array}{c}
\{\ Day(Sunday) \leftarrow, \ldots, Day(Saturday) \leftarrow\ \} \\
\downarrow \\
\{\ \forall x(Day(x) \leftarrow x = Sunday), \ldots, \forall x(Day(x) \leftarrow x = Saturday)\ \} \\
\downarrow \\
\{\ \forall x(Day(x) \leftarrow x = Sunday \vee \cdots \vee x = Saturday)\ \} \\
\downarrow \\
\forall x(Day(x) \Leftrightarrow x = Sunday \vee \cdots \vee x = Saturday)
\end{array}$$

Example 2.3.17. The transformation is well-defined for inductive definitions as well. However, as we will see, the transformation is not equivalence preserving for this type of definition.

$$\begin{array}{c}
\left\{\begin{array}{l} R(A) \leftarrow \\ \forall x(R(x) \leftarrow \exists y(R(y) \wedge G(y,x))) \end{array}\right\} \\
\downarrow \\
\left\{\begin{array}{l} \forall x(R(x) \leftarrow x = A) \\ \forall x(R(x) \leftarrow \exists y(R(y) \wedge G(y,x))) \end{array}\right\} \\
\downarrow \\
\{\ \forall x(R(x) \leftarrow x = A \vee \exists y(R(y) \wedge G(y,x)))\ \} \\
\downarrow \\
\forall x(R(x) \Leftrightarrow x = A \vee \exists y(R(y) \wedge G(y,x)))
\end{array}$$

Exercise 2.3.6. *Apply predicate completion to some of the definitions that we have seen. E.g., that of Uncle or of Instructor.*

Properties of predicate completion Predicate completion transforms every non-inductive FO(ID) definition Δ in an FO explicit definition (see page 50). But $comp(\Delta)$ is also defined for inductive definitions. Does this transformation preserve logical equivalence? Yes, for non-inductive definitions; but not in general for inductive definitions.

Theorem 2.3.1. *If Δ is a non-inductive definition, then Δ and $Comp(\Delta)$ are logically equivalent.*

Theorem 2.3.2. *If Δ is an inductive definition, then Δ logically entails $Comp(\Delta)$ but they are not always equivalent.*

Every model of Δ is a model of $Comp(\Delta)$ but the inverse is not always true.

A smallest counterexample Compare the FO(ID) definition

$$\left\{\begin{array}{l} R(A) \leftarrow \\ \forall x(R(x) \leftarrow \exists y(R(y) \wedge G(y,x))) \end{array}\right\}$$

with its completion.

$$\forall x(R(x) \Leftrightarrow x = A \vee \exists y(R(y) \wedge G(y, x)))$$

Consider the structure $\mathfrak{A}$
- $D_{\mathfrak{A}} = \{1, 2\}$, $A^{\mathfrak{A}} = 1$,
- $G^{\mathfrak{A}} = \{(2, 2)\}$, $R^{\mathfrak{A}} = \{1, 2\}$

What is the reachability set of $1 = A^{\mathfrak{A}}$ in $G^{\mathfrak{A}}$? It is the set $\{1\}$. The induction process derives $1 \in R$ using the base rule and stops. Hence, $\mathfrak{A}$ is not a model of the FO(ID) definition.

But $\mathfrak{A}$ satisfies $comp(\Delta)$. E.g., the following instance is satisfied.

$$\mathfrak{A}[x : 2] \models R(x) \Leftrightarrow x = A \vee \exists y(R(y) \wedge G(y, x))$$

Exercise 2.3.7. *Explain this.*

Exercise 2.3.8. *Use IDP to verify the non-equivalence of the reachability definition and its predicate completion at* `http://dtai.cs.kuleuven.be/krr/idp-ide/?present=ReachCompletion` *Verify that in the incorrect model of the completion, $R(B)$ satisfies the completed definition of R, despite the fact that B is not reachable from A.*

C) Formalizing monotone definitions in SO We characterized the defined set of an inductive definition as the limit of the induction process. In case of *monotone* definitions, the defined set can also be characterized as the *least set satisfying the rules.*

E.g., the set of vertices reachable from A in graph G can be defined also as the least set R of vertices satisfying two conditions:
- A is element of R;
- if $x \in R$ and $(x, y) \in G$ then $y \in R$.

The set defined by a monotone inductive definition is the least set satisfying the material implications that correspond to the rules.

In FO, we cannot express that some set is the least set satisfying a condition. However, in SO we can express this. We saw an example of how to do this on page 61.

Exercise 2.3.9. *Use the technique on page 61 to express in SO that R is the set of reachable vertices from A in G.*

Example 2.3.18. The ways in which we express informal definitions in natural language text, even non-inductive ones, is often more similar to FO(ID) definitions than to equivalences of the kind formalized by explicit definitions in FO.

Confer the erroneous use of the connective $\Leftrightarrow$ on page 51 where we tried to define the set of even numbers by the following equivalence:

$$\forall n(Even(2 \times n) \Leftrightarrow Nat(n))$$

Despite its natural appearance, this formula is erroneous. However, the following very similar (non-inductive) FO(ID) definition correctly defines the set of even numbers in the context of $\mathbb{N}$:

$$\Delta = \{\ \forall n(Even(2 \times n) \leftarrow Nat(n))\ \}$$

Since it is not-inductive, Δ is logically equivalent with its completed definition:

$$\forall m(Even(m) \Leftrightarrow \exists n(m = 2 \times n \wedge Nat(n)))$$

Summary: motivation for adding definitions to FO Definitions are a frequent form of information. In general, some types of inductive definitions cannot be expressed in FO. This will be proven in Chapter 6. This provides the motivation to extend FO with definitions.

The definition construct that was added in FO(ID) is suitable to express the most common forms of definitions of sets found in mathematics: monotone definitions and definitions over some induction order are correctly expressed in FO(ID). This is because its informal semantics is consistent with the idea of iterated safe rule application. Its formal semantics is an extension of well-founded semantics and uses 3-valued techniques.

It is also rule-based, and as such preserves the modularity of definitions frequently found in mathematical and scientific text.

The definitional implication symbol introduced in FO(ID) is another form of conditional, clearly distinguishable from material implication and other forms of conditionals that have been studied.

Quite a few rule-based formalisms exist. E.g., Prolog systems, business rules systems, production rule systems. What these languages have in common is that rules and rule sets as viewed as procedural entities. E.g., in production rule systems, rules are run as bottom up procedures. In Prolog, rules are run in a top down way.

A unique feature of FO(ID) and of the IDP knowledge base system is that rule sets are viewed as expressions of declarative propositions: definitions. They specify information, not procedures. They cannot be run, they cannot be executed, they don't do anything. However, like all information they can be used to solve multiple forms of problems. This point is elaborated in the next paragraph.

Reasoning about definitions A (rule-based) definition expresses a logic relation between a defined concept and the parameters. This relation is that the defined concept is the one that can be constructed by iterated safe rule application.

The strong stress on rule application might have created the impression that a definition is a procedural concept, and iterated rule application is its execution. However, this is not the right way to see it. This is quite clear for non-inductive definitions. E.g., few would argue that the definition of the brother concept is a program to compute brothers from a given value for parameters. Well, the same holds for inductive definitions.

The inference problem with input a definition and values for its parameters, and output the corresponding value of defined predicates, is frequently useful. Systems that perform this form of inference essentially perform iterated rule application (bottom up). However, there are interesting reasoning problems where exactly the inverse is to be done. Where the value of the defined symbol is known partially or completely, and the computational problem is to compute values for the parameters. To solve this problem requires reasoning backward over rules, against the direction of iterated rule application. This shows that definitions are a declarative concept, not bound to a specific sort of execution.

Example 2.3.19. Consider the following FO(ID) theory.

$$\left\{ \begin{array}{l} R(A) \leftarrow \\ \forall x(R(x) \leftarrow \exists y(R(y) \wedge G(y,x))) \end{array} \right\}$$
$$\forall x R(x)$$

This theory imposes the constraint on graph G that every element of the domain should be

reachable by a finite path from A. Consider the computational problem with as input a domain, a value for A in this domain, and the theory itself; as output the value of G in a model of this theory with the given domain and value for A. The value of G in a model is nothing else than a graph such that the reachability set of A in this graph is the entire domain.

A number of computational problems are naturally represented as such. E.g., the problem of laying a cable network from some source to a number of locations to obtain full reachability. Another problem is the *travelling salesman*. Consider a map of cities interconnected by a road relation $Road(x, y)$ with intended interpretation that there is a road from city x to y. Assuming the city A represents the starting point of the traveling salesmen. The problem is to compute a path from A that passes at every city exactly once.

The representation uses $\Sigma = \{A, Road, G\}$. Here, G is the "next" relation of the path. That is, $G(c, c1)$ expresses that $c1$ follows c in the trip. Part of the theory is as follows:

$$\left\{ \begin{array}{l} R(A) \leftarrow \\ \forall x(R(x) \leftarrow \exists y(R(y) \wedge G(y, x))) \end{array} \right\}$$
$$\forall x\ R(x)$$
$$\forall x \forall y(G(x, y) \Rightarrow Road(x, y))$$

They express that the trip follows roads and reaches all cities. Extra axioms are needed to express that G starts at A. Moreover, that G is a path. This means that each node has at most one outgoing edge and at most one incoming edge.

Exercise 2.3.10. *Complete this specification, implement your solution in IDP and check if it is correct for some examples.*

Exercise 2.3.11. *Use IDP to compute all graphs G with total reachability set with domain $\{a, b\}$ and value $A^{\mathfrak{A}} = a$ at* `http://dtai.cs.kuleuven.be/krr/idp-ide/?present=ReachabilityIsTotal`. *Next, extend the structure with value $G^{\mathfrak{A}} = \{(a, a), (b, b)\}$ and try again to find models. Does it give what you expected?*

More examples

Exercise 2.3.12. *Express that binary relation T is the symmetric closure of binary relation G using an inductive definition. We saw already how to express this in SO.*

Example 2.3.20. Given a graph $\rightarrow$, we call a node s *terminating* if there is no infinite path $s \rightarrow s_1 \rightarrow s_2 \rightarrow \ldots$. We represent the graph by the binary predicate symbol $G/2$ and the set of terminating nodes by the unary relation symbol $A/1$. The latter predicate can be defined as follows:

$$\left\{ \ \forall x(A(x) \leftarrow \forall y(G(x, y) \Rightarrow A(y))) \ \right\}$$

This is a monotone definition that uses a universal quantifier in the body.

Let us investigate it in more detail. The antecedent of the rule is trivially satisfied for any x that has no outgoing edge. The set of those nodes forms our first layer of terminating states. The next layer are those states that have only outgoing edges to nodes of the first layer. And so on.

Exercise 2.3.13. *Define the predicate nonterminating/1 as the class of non-terminating states of $\mathcal{M}$. You may introduce as many auxiliary symbols as you need.*

Example 2.3.21. Consider the following simplified access control scenario with a group of users and a file. The file has owner O. A user can delegate access to the file to another user, or can block access for another user. A user x has access if he is the owner or there is a chain of users starting from O and ending with u, each having access and delegating to the next in the chain, and u is not blocked access by any user with access.

$$\left\{ \begin{array}{l} access(O) \leftarrow \\ \forall x(access(x) \leftarrow \exists y(access(y) \wedge delegates(y,x)) \wedge \neg \exists z(access(y) \wedge blocks(z,x))) \end{array} \right\}$$

This is an interesting definition. It is not monotone due to the second condition of the inductive rule. If the blocks relation is empty, it defines *access* as the reachability set from O in the graph *delegates*. If *blocks* is nonempty but there is a hierarchy amongst users with O at the top and each delegation from some user to somebody of the same or lower level, while each blocking is from a user to somebody of strictly lower level, then the definition cleanly defines the access relation which can be computed by iterated safe rule application. However, the definition has also *paradoxical* instances. E.g., assume that owner delegates to A who blocks himself. In that case, A has access (thanks to the owners delegation) unless it has access. Or assume that the owner delegates to A and B and both block eachother. One can play with this problem at `http://dtai.cs.kuleuven.be/krr/idp-ide/?present=access`

The notion of definition is a complex concept that sometimes leads to paradoxes,. Such paradoxes are always caused by the concepts being defined nonmonotically in terms of itself as is the case in the above example. Such paradoxes appear also in text and they can be very subtle. E.g., consider the following definition: *Berry's number is the smallest number not definable in less than 14 words.* But this definition contains less than 14 words? Here in this case, the paradox is due to the use of the concept of "definable". The definition of Berry's number in a true sense adds a case to the definition of what the definable numbers are, and at the same time, uses the definability relation in a non-monotonic way. This is what causes the paradox. `https://en.wikipedia.org/wiki/Berry_parad`

The simplest FO(ID) definition expressing a paradox is:

$$\left\{ \; p \leftarrow \neg p \; \right\}$$

This rule set has no process of iterated safe rule application that leads to saturation. Starting from $\emptyset$, the rule can be applied, but not safely since applying it destroys its own antecedent.

In comparison, the following definition is not a paradox.

$$\left\{ \; p \leftarrow p \; \right\}$$

It allows for a trivial induction process of iterated safe rule applications leading to a saturated set, namely $\emptyset$, the propositional structure in which p is false. Hence, this definition defines p to be false. It is a sensible definition although there are easier ways to define p.

Some familiar looking examples

$$\left\{ \begin{array}{l} \forall x \forall t(Member(x, .(x,t)) \leftarrow \mathbf{t}) \\ \forall x \forall h \forall t(Member(x, .(h,t)) \leftarrow Member(x,t)) \end{array} \right\}$$

$$\left\{ \begin{array}{l} \forall l(Append(Nil, l, l) \leftarrow \mathbf{t}) \\ \forall h \forall t \forall l \forall t_1(Append(.(h,t), l, .(h,t_1)) \leftarrow Append(t, l, t_1)) \end{array} \right\}$$

Do these look familiar to you? They are prototypical Prolog programs written in FO(ID) definitions.

2.3.6 Relation to Prolog

Prolog is a declarative programming language. Its name comes from "programmation en logique". Prolog was invented by Robert Kowalski (Imperial College London) and Alain Colmerauer (U.Marseille) around 1971-1975. A Prolog interpreter is called by posing a query. Prolog has a a procedural semantics: topdown execution of queries, induced by the inference algorithm called SLDNF resolution (Selective Linear Definite clause resolution with Negation as Failure). Special features are unification, backtracking and negation by failure.

Side-effects (cut !, assert and retract,..) make it impossible to view the full Prolog as a declarative (modelling) logic. The subformalism of Prolog without these procedures with side effects is called *Logic Programming*. An example of a logic program is:

Example 2.3.22.

```
parent_child(trude, sally).
parent_child(tom, sally).
parent_child(tom, erica).
parent_child(mike, tom).

female(trude).

sibling(X, Y) :- parent_child(Z, X), parent_child(Z, Y).

father_child(X, Y) :- parent_child(X, Y), not female(X).
mother_child(X, Y) :- parent_child(X, Y), female(X).
```

Queries:

```
?- father_child(mike,tom).
True
?- father_child(tom,X).
X=sally
X=erica
?- mother_child(mike,tom)
False
```

The declarative semantics of logic programming has been a topic of intense research. Originally, a logic program was viewed as a set of Horn clauses: simple material implications. However, this cannot explain Prolog's answers. E.g., the set of atoms and material implications that correspond to the rules in the Prolog example above does not entail `father_child(mike,tom)` since it does not entail the antecedent $\neg$ `female(mike)`.

Exercise 2.3.14. *Explain this.*

During 1975-90, an intensive search for alternative explanation took place. Many formal semantics were defined but no satisfactory informal semantics. One of the early proposals by Keith Clark in 1978 was to view logic programs as definitions. However, his formal semantics was

based on the completion semantics in FO. As we know, this semantics is too weak to express inductive definitions and soon his ideas were rejected. Starting in 1998, I have argued that logic programs under the well-founded semantics can be understood as a logic of (inductive) definitions. The definition construct of FO(ID) is inspired by this.

In this view, the informal semantics of a logic program is then given by:
- UNA and Domain Closure Axiom for the set of all function and object symbols of the program.
- The set of rules interpreted as a definition defining all predicate symbols. If no rule exists for a predicate, it is defined to be the empty relation. E.g. if we delete the unique rule for `female`, female is defined to be empty .
- Negation as failure `not P` is classical negation $\neg P$.
- The rule operator $:-$ is not material implication but definitional implication.

This view explains the answers to all queries in the example. It works for recursive programs and gives a natural and precise informal semantics to many logic programs. Moreover, it explains the meaning of the negation as failure connective in a simple and satisfactory way: it is classical negation.

Exercise 2.3.15. *UNA is needed. Show that the example logic program on page 77 viewed as a definition in FO(ID) without UNA, does not entail* `father_child(mike,tom)`. *You need to sketch a structure that satisfies the definition (but not UNA), in which this atom is false.*

Relationship logic programs - FO(ID) As a modelling language, logic programming viewed as a logic of definitions is a poor language because incomplete knowledge cannot be expressed in it. Indeed, logic programs are categorical. They have a unique model; hence, no incomplete knowledge can be represented in Prolog. In the context of modelling and knowledge representation, one almost always has to deal with incomplete knowledge.

FO(ID) arose as an integration of Logic Programming with FO, with the aim of combining the strengths of both. In the first place, it extends the notion of definition by dropping the condition that "all" predicates are defined. To this end FO(ID) definitions define only the predicates appearing in the head of rules. Parameter symbols e.g., are not defined. This makes the notion of FO(ID) definition suitable for expressing incomplete knowledge. FO(ID) theories extend logic programs in the following ways:
- UNA and DCA are absent in FO(ID), but can be expressed in the language.
- Definitions define only the predicates in the head of rules, no other symbols.
- Multiple definitions are allowed.
- FO and FO(.) axioms
- Aggregates

Summary This section was concerned with the question: if we want to see logic programming as a modelling language, what information does a program express? The answer that I gave, improving on Clark's thesis is that a "logic program=(inductive) definition". It is this view that is further developed in FO(ID,...) and is implemented in the IDP system.

Another field based on a declarative view of logic programming is the field of *Answer Set Programming*.

2.3.7 FO(.)

It custom to denote extensions of FO as F(<ext>) with <ext> the extension. This section introduced extensions FO(Types), FO(Types,Arit), FO(ID), FO(Agg), and combinations of it.

Strictly speaking, with FO(.), we mean the family of logics obtained by extending FO. The system IDP supports one such a language, which corresponds more or less to FO(Types,Arit,ID,Agg). In this course, by small abuse of terminology, we use FO(.) to denote this extension of FO.

2.4 Two examples

Weekly steel oven scheduling

> Assume we have three types of resources: 3 platforms, 2 ovens, and 1 cooler. There are a number of tasks (of constructing one or more steel objects). Each task uses a single platform for: one hour loading, then some hours of oven, and then five hours cooling, one hour deloading. We want to do 20 tasks in the minimal amount of time: nine of them need 10 hours of oven, five need 12h, one needs 15h, two need 16h, and three need 22h.

We start with the design of a suitable vocabulary. There are certainly many ways to do this. One option is as follows.

- The following types are relevant: $Platform, Oven, Cooler, Task, Time$.
  - $Time$ is *int* or a subset of it.
  - $Cooler$: we have only one cooler. Types with only one element can always be eliminated. The effect on predicate and function symbols is that the argument of this type is dropped and becomes implicit. The advantage is simplicity and efficient reasoning. The disadvantage is on the level of robustness and extensibility: if later coolers are added, the theory needs to be rewritten. So, we keep the type $Cooler$.
- Durations: the load, bake and cool operations of a task have durations. Only the duration of the baking phase varies from task to task. We choose a function symbol to represent this: $dur(Task) : Time$. We leave the durations of the load and cool phase implicit. The disadvantage of this is a lower robustness of our theory: if new tasks show up that need more time to load than others, the theory must be extended with a duration function for the loading phase of tasks.
- A task is to be assigned 1 platform, 1 oven, 1 cooler, each during a certain period. Since only one resource of each type is assigned to the task, we mauy choose function symbols to represent the resources assigned to the task:
  $PlOf(Task) : Platform, OvOf(Task) : Oven, CoOf(Task) : Cooler$
  The intended interpretation of these symbols is obvious.
- Each phase of the execution of a task is characterised by a start and stop time; these are 6 time points. We choose 6 function symbols with the obvious intended interpretation:
  $StartPl(Task) : Time, StopPl(Task) : Time, StartOv(Task) : Time, StopOv(Task) : Time, StartCo(Task) : Time, StopCo(Task) : Time$.

- Finally, object symbols can be used to represent the objects. In our case, 20 tasks $o1, \ldots, o20$, 3 platforms $pl1, pl2, pl3$, 2 ovens $ov1, ov2$, 1 cooler co. However, in IDP, we can better specify these as domain elements inside a structure.

Summarized, the vocabulary Σ is as follows:

- Types $Platform, Oven, Cooler, Task, Time$.
- $Dur(Task) : Time$ - the duration of oven time of tasks (all other durations have the same
- $PlOf(Task) : Platform, OvOf(Task) : Oven, CoO(Task) : Cooler$
- $StartPl, StopPl, StartOv, StopOv, StartCo, StopCo : Task \to Time$
- Constants for tasks $o1, \ldots, o20$, platforms $pl1, pl2, pl3$, ovens $ov1, ov2$, and cooler co.

The theory consists of the following laws:

- The temporal links between different phases of tasks.

$$\begin{aligned}\forall o[Task](\ & StartOv(o) = StartPl(o) + 1 \wedge \\ & StopOv(o) = StartOv(o) + Dur(o) - 1 \wedge \\ & StartCo(o) = StopOv(o) + 1 \wedge \\ & StopCo(o) = StartCo(o) + 4 \wedge \\ & StopPl(o) = StopCo(o) + 1)\end{aligned}$$

- No simultaneous usage of platforms, ovens, coolers:

$$\begin{aligned}\forall pl\ & \forall o1 \forall o2 (pl = PlOf(o1) \wedge pl = PlOf(o2) \\ & \Rightarrow StopPl(o1) < StartPl(o2) \vee StartPl(o1) > StopPl(o2)) \\ \forall ov\ & \forall o1 \forall o2 (ov = OvOf(o1) \wedge ov = OvOf(o2) \\ & \Rightarrow StopOv(o1) < StartOv(o2) \vee StartOv(o1) > StopOv(o2)) \\ \forall co\ & \forall o1 \forall o2 (co = CoOf(o1) \wedge co = CoOf(o2) \\ & \Rightarrow StopCo(o1) < StartCo(o2) \vee StartCo(o1) > StopCo(o2))\end{aligned}$$

- We can enumerate the types of objects, platforms, coolers in the following axioms:

$$\begin{aligned}& \forall x[Task](x = o1 \vee x = o2 \vee \ldots x = o20) \\ & \forall x[Platform](x = pl1 \vee x = pl2 \vee x = pl3) \\ & \forall x[Cooler]x = co1 \\ & \forall x[Oven](x = ov1 \vee x = ov2)\end{aligned}$$

  Furthermore, we need to specify UNA axiom for tasks, platforms, and ovens:

$$\neg(o1 = o2), \ldots$$

  and the durations of tasks:

$$Dur(o1) = 5, \ldots$$

  Alternatively, in IDP these data can be put in an input structure.

There are other options for the vocabulary. E.g., use a predicate symbol

$$Assign(Task, Resource, Time)$$

where *Resource* is the supertype of platforms, ovens, and coolers. Its intended interpretation is:

- $Assign(o, r, t)$: resource r is assigned to task o at time t

Such a choice would have a strong impact on the theory. Under come circumstances, the resulting theory would be more robust. E.g., it would be easier to express that more than one platform, oven, or cooler is needed for a task. That is impossible in the original theory due to the use of function symbols that describe the resource for a task. It would be easier to introduce other types of resources.

In general, the choice of the vocabulary (the ontology) has a strong impact on the theory, its robustness, its extensibility and last but not least, the efficiency by which solvers can solve the relevant tasks.

Exercise 2.4.1. *Elaborate the steel oven problem using this symbol and potentially others you find useful.*

Exercise 2.4.2. *What form of inference do we need to apply on this theory (and potentially other input objects) to compute a correct schedule for a given steel oven problem?*

Modelling transition structures

Transition structures are abstract representations of dynamic systems. They are state based and capture all potential evolutions of systems at once by describing the possible transitions between states. They are similar to finite state machines.

Definition 2.4.1. A *transition structure* $\mathcal{M}$ for propositional vocabulary σ is a triple $\langle S, \rightarrow, L\rangle$ where S is a set of states, $\rightarrow$ is the binary *transition relation* on S, and L is a mapping $S \rightarrow 2^{\sigma}$, where σ is a vocabulary of proposiontal symbols, and for each state s, $L(s)$ is a subset of σ.

Thus, elements of S represent states. The state of affairs at some state $s \in S$ is represented by $L(s)$, a set of propositional symbols representing those that are true in state s. Notice that $L(s)$ can be viewed as a propositional structure, a set of true propositional properties. It represents a snapshot of the world at state s.

We first introduce a logical vocabulary Σ_{ts} to express a transition structure as a Σ_{ts}-structure in FO. Next, we discuss a number of formulas that express temporal propositions about the world modelled by the transition system.

Given a transition model $\mathcal{M} = \langle S, \rightarrow, L\rangle$, where $L(s)$ is a set op propositional symbols of σ. We choose Σ_{ts} to consist of the binary predicat $T/2$ and symbols $P/1$, for every $P \in \sigma$:

- Types: there is only one type which is the type of states. Hence, we can use untyped logic.
- A binary predicate $T/2$ ("T" for Transition). Its value/interpretation in a voc_{ts}-structure is the transition graph between states.
- State unary predicates: for every propositional symbol $P \in \sigma$, we introduce a unary predicate symbol $P/1$. Its value/interpretation in a structure is the set of states s such that $P \in L(s)$, i.e., P is true in s.

Under this choice of vocabulary, a transition model $\mathcal{M}$ corresponds to a unique Σ-structure $\mathfrak{A}_{\mathcal{M}}$:
– $D_{\mathfrak{A}_{\mathcal{M}}} = S$;

- $T^{\mathfrak{A}_{\mathcal{M}}} = \rightarrow$; i.e., $T^{\mathfrak{A}_{\mathcal{M}}} = \{(x, y) \mid x \rightarrow y\}$;
- for each $P \in \sigma$, $P^{\mathfrak{A}_{\mathcal{M}}} = \{s | P \in L(s)\}$.

In this vocabulary, we can not express properties of the world. What is the meaning of the following formulas?

- $\forall x \forall y \forall z (T(x, y) \wedge T(x, z) \Rightarrow y = z)$: transitions are deterministic: each state has at most one successive state.
- $\forall x \exists y T(x, y)$: viewing a state without successor state as a deadlock state, this proposition expresses that the transition structure is deadlock free.
- $\exists x (Wait(x) \wedge \forall y (\neg Wait(y) \Rightarrow T(y, x)))$: there exists a wait state such that each non-wait state there is a transition to this wait-state.
- $\exists x (Wait(x) \wedge \forall y (Wait(y) \Rightarrow y = x))$: the wait state is unique.

2.5 Axiomatizing some structures

2.5.1 Axiomatizing the information in a database

A database DB contains information about the application domain. What is this information? Here we investigate this question in the following way: we build an FO (or FO(.)) theory $\mathtt{Th}(DB)$ that contains the same information, and that in principle could be used to solve queries in the same way as the original database.

A database from a logic perspective A database instance DB contains a set of tables, each consisting of a set of cells on the intersection of rows and columns. Each cell contains a string or a number. Database systems solve queries on such database instances. For simplicity, we assume here that each value in a cell is a string, and not a number.

The closest thing in logic to a database instance DB is a Herbrand structure. Let S_{DB} be the set of strings in the cells. Define the vocabulary Σ_{DB} as the set of object symbols S_{DB} augmented with one predicate symbol P/n for every table with n columns. DB then corresponds to the Herbrand structure $\mathfrak{A}_{DB}$ over Σ_{DB}:

- its domain $D_{\mathfrak{A}_{DB}}$ is S_{DB},
- for each object symbol $C \in S_{DB}, C^{\mathfrak{A}_{DB}} = C$,
- for each predicate symbol $P \in \Sigma_{DB}$, $P^{\mathfrak{A}_{DB}}$ is the set of tuples of object symbols corresponding to the rows in the table associated with P.

Queries over the database are normally phrased in SQL. In the context of logic, queries correspond to sentences ϕ and set expressions $\{\bar{x} : \phi[\bar{[}x]\}$ over Σ_{DB}. The answer to a query is the value $\phi^{\mathfrak{A}_{DB}}$ or $\{\bar{x} : \phi[\bar{[}x]\}^{\mathfrak{A}_{DB}}$ of the query in $\mathfrak{A}_{DB}$. Thus, in this logical perspective on databases, query problems of databases correspond to instances of the evaluation inference problem: taking as input a structure $\mathfrak{A}$ and an expression e and returning as output the value $e^{\mathfrak{A}}$ of the expression.

Example 2.5.1. The following is a simple database instance in the application field of the (fictious) master informatics at the university of Lilliput. It stores the current courses, lecturers, students and grades.

Instructor

Ray	CS230
Hec	CS230
Sue	M100
Sue	M200

PassingGrade

AAA
AA
A
B
C

Enrolled

Jill	CS230
Jack	CS230
Flo	CS230
Jill	M100
Flo	M100
Jill	M200
Flo	M200

Prerequ

CS230	M100
M100	M200

Grade

Jill	CS230	A
Jack	C230	C
Flo	CS230	AA
Jill	M100	D
Flo	M100	A
Flo	M200	AAA

We can phrase "queries" over this structure using logic sentences ϕ and set expressions e over Σ_{DB}. The answer to a query is given by its value in $\mathfrak{A}_{DB}$.

- Take the query "somebody teaches more than one course". It is expressed by the following sentence ϕ:

  $$\exists x \exists y \exists z (Instructor(x, y) \wedge Instructor(x, z) \wedge y \neq z)$$

  It holds that $\phi^{\mathfrak{A}_{DB}} = \mathbf{t}$ since Sue teaches two course M100 and M200.

- Take the query for "the set of students that succeeded for all courses in which they registered". It is formalized by the following set expression e:

  $$\{s : \forall c (Registered(s, c) \Rightarrow \exists g (Grade(s, c, g) \wedge PassingGrade(g)))\}$$

  One can verify that the value $e^{\mathfrak{A}_{DB}}$ is $\{Flo\}$.

A database instance DB is a precise formal object, and so is the Herbrand structure $\mathfrak{A}_{DB}$. But a database is also intended as a modelling of a application domain. In the mind of the users, object symbols c in tables have an informal interpretation $\mathcal{I}(c)$ as entities in the application domain. E.g., in the running example, $\mathcal{I}(Ray)$ is Prof. Ray Reiter; $\mathcal{I}(CS230)$ is the course "Logic for computer science", ... Table names P have an informal interpretation as relational concepts $\mathcal{I}(P)$ in the application domain: e.g., $\mathcal{I}(Enrolled)$ is the relational concept of students being enrolled in a course. The Herbrand structure $\mathfrak{A}_{DB}$ is an abstraction of a unique state of affairs $\mathcal{I}(\mathfrak{A}_{DB})$. A database is meant to provide detailed information about the actual state of affairs of the application domain. What is this information exactly? That question is not as trivial as may seem.

The question of the information that users extract from DB more or less corresponds to the question of what users consider to be possible states of affairs of the application domain. This depends on DB and $\mathcal{I}$ but it also depends on meta-knowledge in the mind of the user: meta-knowledge about the correctness and the completeness of the tables of DB. A user who believes that all tables are correct (its rows represent true facts of the actual state of affairs) and complete (true facts of the actual state of affairs are represented by a row in the table), will assume that the actual state of affairs is $\mathcal{I}(\mathfrak{A}_{DB})$, and that this is the only possible state of affairs of the application domain. In this case, DB represents complete knowledge on the actual state of affairs. Queries posed to DB return correct and complete answers.

On the other hand, if the user believes that certain tables are incomplete or worse, incorrect, then $\mathfrak{A}_{DB}$ is not the only possible state of affairs according to the user. It happens often enough

that tables are known to be incomplete. E.g., in the Lilliput database, imagine the user may consider it possible that the tuple $(Jack, M100)$ is missing in the Enrolled table. In a company, the database may contain an incomplete table recording producers of certain chemical products. Queries on this table may return incomplete or even wrong answers. E.g., the query for all companies in a range of 10km that do not produce ammonia, will return a wrong answer if there is an ammonia producing company closer than 10km which is not recorded in the producer table. In the current state of the art, the meta-information about completeness and incorrectness of tables is a responsbility of the user. It is not stored in the database and the database system cannot warn the user about potential incompleteness or incorrectness of query answers. It is a weakness of state of the art database systems.

Exercise 2.5.1. *Assume the user knows that a hacker eliminated all data from the course Philosophy (Ph100) from the student database. How does this affect his interpretation of the answer to the two queries of Example 2.5.1?*

In what follows we will assume that DB is correct and complete. In the terminology of modelling, the actual state of affairs is $\mathcal{I}(\mathfrak{A}_{DB})$, the state abstracted by DB. No other state is possible. Under this meta-assumption, there is complete knowledge of the actual state of affairs and query answers returned by the database system are correct and complete.

Now, we capture the information in the DB as a logic theory $\mathtt{Th}(DB)$. This is interesting for several reasons: 1) to understand the implicatures hiding below a database, 2) to learn how to make common implicatures explicit in FO and 3) to understand what information can be represented in a database and, also of interest, what information cannot be expressed in it.

A first attempt at formalization in FO would be to express the database as a conjunction ϕ_{DB} of all facts in the tables:

$$Instructor(Ray, CS230) \wedge \cdots \wedge Passinggrade(AAA) \wedge \cdots \wedge Enrolled(Jill, CS230) \wedge \cdots \wedge Prerequ(CS230, M100) \wedge \cdots \wedge Grade(Jill, CS230, A) \wedge \cdots \wedge Grade(Flo, M200, AAA)$$

Is this correct? Certainly not. This can be seen in different ways. First of all, the class of models of ϕ_{DB} is broad and contains many structures that are clearly not abstractions of possible states of affairs of the application domain. E.g., for the running example, take the following structure $\mathfrak{A}$ with domain $\{a\}$, such that:

- $c^{\mathfrak{A}} = a$
- $PassingGrade^{\mathfrak{A}} = \{a\}$,
- $Enrolled^{\mathfrak{A}} = Instructor^{\mathfrak{A}} = Prerequ^{\mathfrak{A}} = \{(a, a)\}$,
- $Grade^{\mathfrak{A}} = \{(a, a, a)\}$.

In this structure, only one object exists, a, and all symbols denote that same object. The object a is enrolled in itself, a is a passing grade, ... This is certainly not an abstraction of the actual state of affairs of the application domain as intended by DB. But it is a model of ϕ_{DB}. Thus, this theory is not a correct modelling.

Exercise 2.5.2. *Show that this structure $\mathfrak{A}$ satisfies ϕ_{DB}.*

There is a second way in which we can see that this axiom ϕ_{DB} does not express the information in the database: . From the previous paragraph, we know that ϕ_{DB} allows far more possible states of affairs than DB, hence, it contains far less information than DB. One would expect then that in many cases, ϕ_{DB} does not entail the answers to queries that a database system would return. This is indeed the case.

- ϕ_{DB} does not entail $\neg(Ray = AAA)$. This follows from the above model while a database system would certainly answer "yes" to the query $\neg(Ray = AAA)$.
- The axiom does not entail $\neg Enrolled(Jack, M200)$, while a database system answers "yes" to the query $\neg Enrolled(Jack, M200)$.

Exercise 2.5.3. *Prove that ϕ_{DB} indeed does not entail these answers. (Hint: find a model of ϕ_{DB} in which the propositions $\neg(Ray = AAA)$ and $\neg Enrolled(AAA, CS230)$ are false. Do not look too far.)*

Our first attempt to express the information in DB failed miserably. What we learn from it is that beyond the conjunctive formula ϕ_{DB} which express all the positive information in the database tables, a database contains a number of implicatures. In fact, we have met similar implicatures earlier on. They are summed up in the following definition.

Definition 2.5.1. Given a database DB with vocabulary Σ_{DB}, and set $S_{DB} = \{C_1, \ldots, C_n\}$ of object symbols occurring in the cells of DB. Define $\texttt{Th}(DB)$ as the theory consisting of the following axioms:

- $UNA(S_{DB})$:
$$\{\neg(C_i = C_j) | C_i, C_j \text{ are different symbols in } S_{DB}\}$$
- $DCA(S_{DB})$:
$$\forall x(x = C_1 \vee x = C_2 \vee \cdots \vee x = C_n)$$
- $Comp(P)$: the completed definitions of the table predicates P/n of DB (see page 53).

Example 2.5.2. In the above example database, the completed definitions of predicate table of $Instructor$ has the form:

$$\forall x \forall y(Instructor(x, y) \Leftrightarrow (x = Ray \wedge y = CS230) \vee \cdots \vee (x = Sue \wedge y = M200))$$

Alternatively, we could use a definition in FO(ID):

$$\left\{ \begin{array}{l} Instructor(Ray, CS238) \leftarrow \\ \ldots \\ Instructor(Pat, CS238) \leftarrow \end{array} \right\}$$

Some explanation:

- $UNA(S_{DB})$ is the theory consisting of the disequality atoms $\neg(C_i = C_j)$. It expresses that different object symbols represent different entities in actual state of affairs. Thus, a database designer should not use more than one symbol to denote the same entity.
- $DCA(S_{DB})$ expresses that no other entities than those referred to in the database exist in the actual state of affairs. Thus, a database designer should introduce a symbol for each entity in the actual state of affairs.
- $Comp(P)$ expresses that the atomic facts occurring in DB are true and those not in the database are false in the actual state of affairs.

Now, we argue that $\texttt{Th}(DB)$ indeed captures all information in DB about the application domain.

Theorem 2.5.1. *For every Σ_{DB}-structure $\mathfrak{A}$, $\mathfrak{A} \models \texttt{Th}(DB)$ iff $\mathfrak{A}$ is isomorphic to the Herbrand structure $\mathfrak{A}_{DB}$.*

Proof. Sketch. Let $\mathfrak{A}$ be a model of $\texttt{Th}(DB)$. Consider the function $b : S_{DB} \to D_{\mathfrak{A}} : c \mapsto c^{\mathfrak{A}}$. Note that $S_{DB} = D_{\mathfrak{A}_{DB}}$. We show that b is an isomorphism from $\mathfrak{A}_{DB}$ to $\mathfrak{A}$. Since UNA is satisfied by $\mathfrak{A}$, b is injective (it maps different object symbols to different domain elements). Since DCA is satisfied by $\mathfrak{A}$, b is surjective (for each $d \in D_{\mathfrak{A}}$, there is a $c \in S_{DB}$ such that $b(c) = d$). Hence, b is a bijection. Finally, because $\mathfrak{A}$ satisfies $comp(P)$ for each predicate symbol P/n, it follows that for any tuple $(c_1, \ldots, c_n)$ of object symbols, it holds that $(c_1, \ldots, c_n) \in P^{\mathfrak{A}_{DB}}$ iff $({c_1}^{\mathfrak{A}}, \ldots, {c_n}^{\mathfrak{A}}) \in P^{\mathfrak{A}}$. Hence, b is an isomorphism. ■

This theorem shows that $\texttt{Th}(DB)$ fully "knows" the actual state of affairs. Indeed, all its models are abstractions of it. It also entails that $\texttt{Th}(DB)$ entails the answer to each query on DB. Indeed, the isomorphism Theorem 2.2.1 entails for each sentence φ over Σ_{DB}, that φ is true in the Herbrand structure $\mathfrak{A}_{DB}$ iff $\texttt{Th}(DB)$ logically entails φ. Hence, the theory $\texttt{Th}(DB)$ contains all information of the database.

In principle, a theorem prover called to prove $\texttt{Th}(DB) \models \varphi$ will return the same answer as a database system would return when called to evaluate whether φ is true in DB. In a sense, it means that the database query procedure can be seen as a special purpose theorem prover to reason about theories $\texttt{Th}(DB)$. In practice, theorem provers lack efficiency to compute answers from $\texttt{Th}(DB)$ for large enough DB.

The theory $\texttt{Th}(DB)$ has some special properties. $\texttt{Th}(DB)$ is categorical: it admits exactly one model modulo isomorphism. This entails that it represents *complete knowledge*: for each formula φ over Σ_{DB}, $\texttt{Th}(DB)$ entails φ or $\texttt{Th}(DB)$ entails $\neg\varphi$. Hence, $\texttt{Th}(DB)$ "knows" the answer to each query. This is very uncommon in many modelling applications. Typically knowledge is incomplete. There is missing information. There are multiple possible states of affairs. But this is not the case with databases.

From a modelling point of view, this section shows that it is a little cumbersome to express complete knowledge in FO: UNA and $comp(P)$ are long and tedious. The same holds for DCA, which moreover cannot always be expressed in FO, as discussed later in the course. By extending FO with definitions, the situation improves: e.g., the information in tables, to be represented in FO by $comp(P)$, can be represented in a modular, table-like format as a definition by exhaustive enumeration. E.g., $comp(Prerequ)$ can be represented as the definition

$$\left\{ \begin{array}{l} Prerequ(CS230, M100) \leftarrow \\ Prerequ(M100, M200) \leftarrow \end{array} \right\}$$

But the inverse holds as well: the above theorem shows that databases are unsuitable to represent incomplete knowledge. E.g., in the running example, assume that it is unknown whether *Jack* is enrolled for course $M100$. Now, there are multiple possible states of affairs. This knowledge cannot be represented in the database (or at least, not in the sort of database that is considered here). In many modelling applications, only incomplete knowledge is available, and databases are useless then.

Exercise 2.5.4. *For each component: UNA, DCA or comp(P), of* $\mathtt{Th}(DB)$ *), give a query Q such that Q's answer is not longer entailed if we drop that component and prove your answer.*

We end this section with some remarks on the logical perspective on databases.

Remark 2.5.1. In the context of databases, it is common to use tables with large arity. They are used to represent complex objects together with all their attributes. E.g., a table of arity 25 with rows representing people and their first name, last name, date of birth, address, ... In such a case (and only in such a case in my opinion), the logical representation of a query can be very ugly. E.g., a query to find first and family name of all persons born since 2000:

$$\{(fn, famn) : \exists x_3 \ldots \exists x_{25}(Person(fn, famn, x_3, \ldots, x_{25}) \wedge (\text{`}1/1/2000\text{`} \leq x_3)\}$$

For this type of database tables, there exists much more natural translations into logic. They are obtained by splitting up such object relations in primitive basic relations,
e.g., $FirstName(Person, string), LastName(Person, string), DateOfBirth(Person, Date), \ldots$.

Remark 2.5.2. The most problematic axiom of $\mathtt{Th}(DB)$ is $DCA(DB)$, which states that no objects exist except those named in the database. The problem is with types such as numbers and dates. Here $DCA(DB)$ entails that only the numbers or the dates that occur in one of the tables exist. This is obviously not correct. E.g., take the query for the number n of students enrolled in CS230. Suppose that n does not occur in a cell of the database. Can the database return n despite the fact that "n does not exist"? It is clear that $DCA(DB)$ is false: there are many objects that do not appear in the tables but that exist nevertheless.

We can drop $DCA(DB)$ from $\mathtt{Th}(DB)$. The resulting theory has now infinitely models $\mathfrak{A}$ which are very similar: their domains consist of the original named objects and an arbitrary number of additional unnamed objects; the tables $P^{\mathfrak{A}}$ are as before and consists of tuples of named objects. Still, it is not difficult to find queries that cannot longer be answered with this theory. E.g.,

$$\{x : x = x\}$$
$$\{(x, y) : \neg Enrolled(x, y)\}$$
$$\#\{x : x \neq Ray \wedge \cdots \wedge x \neq C\} = 0$$

The value of these expressions depends on the domain of $\mathfrak{A}$, hence, it is different in different models of $\mathtt{Th}(DB) - DCA(DB)$. It follows that a (unique, certain) answer is not entailed by $\mathtt{Th}(DB) - DCA(DB)$. Such expressions are called *domain dependent.*

Exercise 2.5.5. *Show that each of the queries in the preceding paragraph has n value that depends on the domain of the structure* $\mathfrak{A}$ *in which it is evaluated.*

However, there is a class of queries called *safe queries* which are domain independent. Such queries have the same answer in all models of $\mathtt{Th}(DB) - DCA(DB)$. It turns out that SQL imposes syntactic restrictions such that all itsqueries are safe and hence domain independent. Hence, $\mathtt{Th}(DB) - DCA(DB)$ is an acceptable formalization of the semantics of databases, even though it has no unique model.

Exercise 2.5.6. *To give you some insight in safe queries: try to formulate a domain dependent queries in SQL. It should not be possible. Why does it fail?*

Remark 2.5.3. Originally, Codd proposed relational algebra as a query language. Expressions in relational algebra describe a computational process of computing relations through relational operations of intersection, union, join, ... Codd also formulated a declarative view for this

query language: relational algebra queries can be declaratively interpreted as set expressions $\{\bar{x} : \phi[\bar{x}]\}$ with ϕ a FO formula. The execution of the relational algebra term in DB computes the value of the corresponding set expression in the context of $\mathfrak{A}_{DB}$.

Later SQL was much extended and it gradually moved away from this declarative view. Partly, this was motivated by the fact that some useful classes of queries could not be expressed in FO. A few examples: queries for the number of entries in a table, or the sum of numerical values in one column of a table. Notice that in FO(.), we would use aggregate symbols to express such queries. Or queries on inductively definable concepts. E.g., the query for the indirect prerequisites of a course. E.g., in the running example, $CS230$ is a prerequisite of $M100$ which in turn is prerequisite to $M200$, hence $CS230$ is indirect prerequisite to $M200$. In general, such queries cannot be expressed in FO. They can be expressed in SQL using loop constructions. They can be expressed in FO(.) using the declarative concept of inductive definitions, as argued in the section on that topic.

Remark 2.5.4. Some efforts were done to extend databases with some (weak) form of incomplete knowledge. One technique is using null values that represents existing but unknown entities. E.g., to represent that Ray teaches some unknown course, and that the course Ph100 is taught by some unknown lecturer, one can add tuples

$$(Ray, null)$$
$$(null, Ph100)$$

to the instructor table of the student database. Here the interpretation of such a database as a structure is impossible, but we can still define $\mathtt{Th}(DB)$ fairly easily.

Exercise 2.5.7. *To define* $\mathtt{Th}(DB)$ *for databases with this sort of null value, introduce for each occurrence of "null" in a table a separate new object symbol and replace the occurrence of "null" by that object symbol in the table. Thus, the above tuples involving nulls would be replaced by:*

$$(Ray, n_1)$$
$$(n_2, Ph100)$$

in the instructor table. Now, define $\mathtt{Th}(DB)$ *as consisting of* $UNA(S_{DB})$ *and* $comp(P)$ *as defined before, except that UNA is only applied to identifiers, not to the new null object symbols* n_1, n_2*. Argue that this theory is a natural formalization of the meaning of databases with this sort of null values.*

2.5.2 Peano arithmetic

The structure of the natural numbers is a fundamental concept in mathematics and formal sciences. It is used in the next chapter, for modelling *time points*.

The structure $\mathbb{N}$ for $\Sigma = \{0, S/1 :, +/2 :, \times/2 :\}$.
- $D_{\mathbb{N}}$ is the set of natural numbers
- $0^{\mathbb{N}} = 0 \in \mathbb{N}$: the first 0 is a symbol (a numeral), the second is the natural number.
- $S^{\mathbb{N}} : \mathbb{N} \to \mathbb{N} : n \to n + 1$: called the *successor function*.
- $+^{\mathbb{N}}, \times^{\mathbb{N}}$: sum and product functions in $\mathbb{N}$

Formalizing $\mathbb{N}$: can we build a theory $\mathtt{Th}(\mathbb{N})$ such that its unique model is $\mathbb{N}$? More exactly, such that all its models are isomorphic to $\mathbb{N}$? This was studied by Giuseppe Peano, Italian mathematician, "Arithmetices principia, nova methodo exposita", 1889.

– *The Peano axioms* (PA)

This goal is similar as the goal of axiomatizing a database by $\texttt{Th}(DB)$. The theory has a similar structure as $\texttt{Th}(DB)$. It consists of axioms corresponding to UNA, DCA and definitions of $+$ and $\times$. The challenge is that $\mathbb{N}$ is infinite. Can one build a finite theory in FO?

Notational convention The terms $0, S(0), S(S(0)), \ldots$ will be denoted by subscripting S with the number of applications of S. E.g., $S^0(0)$ denotes the term 0, $S^1(0)$ the term $S(0)$, ..., $S^n(0)$ denotes the term $S(\ldots(S(0))\ldots)$ with n occurrences of S. Its value in the structure $\mathbb{N}$ is n, i.e., $(S^n(0))^{\mathbb{N}}$ is n.

The Peano axioms of natural numbers

– $\forall n \neg(0 = S(n)))$
$\forall n \forall m(S(n) = S(m) \Rightarrow n = m)$
The second axiom expresses that S is an injective function. Together they express $UNA(\{0, S\})$: infinitely many disequalities $\neg(S^n(0) = S^m(0)), n \neq m$ represented finitely.

– $\forall n(0 + n = n)$
$\forall n \forall m(S(n) + m = S(n + m))$
These axioms recursively characterize $+$. They are a finite representation of the infinitely many axioms $S^n(0) + S^m(0) = S^{n+m}(0)$, for all $n, m \in \mathbb{N}$.

– $\forall n(0 \times n = 0)$
$\forall n \forall m(S(n) \times m = m + (n \times m))$
These axioms characterize $\times$: a finite representation of the infinitely many axioms of the form $S^n(0) \times S^m(0) = S^{n \times m}(0)$.

What remains to be expressed is that every element in the domain is represented by one of the terms $0, S(0), S(S(0)), \ldots, S^n(0), \ldots$. This corresponds to the *domain closure axiom* as seen in $\texttt{Th}(DB)$. It is to represent the domain closure for the set of terms of the vocabulary $\{0, S\}$. Intuitively, it would be expressed as an infinite "formula":

$$\forall x(x = 0 \vee x = S(0) \vee x = S(S(0)) \vee \ldots)$$

Unfortunately, this is not a formula.

Expressing domain closure for $\{0, S\}$ Idea: the set of natural numbers is the set of objects reachable from 0 through application of the successor function.

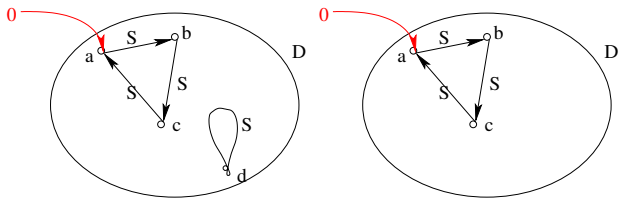
The mathematical way for expressing this:

Definition 2.5.2. The set of natural numbers is defined inductively:
- 0 is a natural number.
- if n is a natural number then the successor of n is a natural number.

Following earlier terminology, the set of natural numbers is the reachability set of 0 in the (graph of the) successor function.

It is one of these inductive definitions that cannot be expressed in FO (see chapter 7) but can be expressed in SO and in languages with inductive definitions including FO(ID).

Example 2.5.3. Two $\{0, S\}$-structures $\mathfrak{A}_1, \mathfrak{A}_2$ in graphical form:

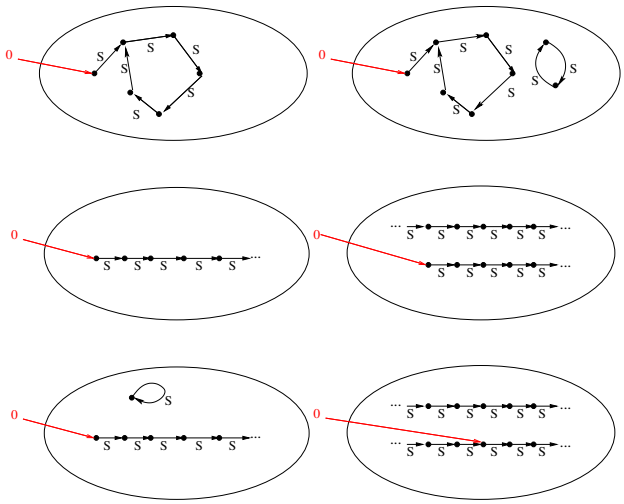


For both $\mathfrak{A} \in \{\mathfrak{A}_1, \mathfrak{A}_2\}$, the set $\{0^{\mathfrak{A}}, (S(0))^{\mathfrak{A}}, (S^2(0))^{\mathfrak{A}}, (S^3(0))^{\mathfrak{A}}, \dots\}$ is the set $\{a, b, c\}$. It is the reachability set of $0^{\mathfrak{A}}$ in $S^{\mathfrak{A}}$. $\mathfrak{A}_1$ does not satisfy domain closure because of d. $\mathfrak{A}_2$ satisfies the domain closure.

$\mathfrak{A}_2$ does not satisfy $UNA(\{0, S\})$. In particular, the following instance of the first UNA-axiom is not satisfied:

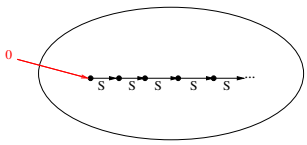
$$\mathfrak{A}_2[n : c] \not\models \neg(0 = S(n))$$

Exercise 2.5.8. *In which is the domain closure satisfied? In which of these UNA is satisfied? What UNA-instances are violated?*



(Some domains are infinite, unbounded at the right and/or unbounded at the left.)

The only topology that satisfies both domain closure and UNA is:



Expressing domain closure in SO Peano expressed domain closure through a sentence of *second order logic*:

$$\forall N(N(0) \wedge \forall x(N(x) \Rightarrow N(S(x))) \Rightarrow \forall x N(x))$$

"Each set containing 0 and closed under taking successor contains all domain elements"

This SO axiom is called *Peano's induction axiom.*

Expressing domain closure in FO(ID) Using an auxiliary symbol $N/1$, we can represent the induction axiom as the following FO(ID) theory:

$$\left\{ \begin{array}{l} N(0) \leftarrow \\ \forall n(N(S(n)) \leftarrow N(n)) \end{array} \right\}$$
$$\forall n N(n)$$

This theory expresses:
- N is defined as the reachability set of 0 in S.
- N contains all elements of the domain.

Together, it is stated here that the domain is the reachability set of 0 in S.

Peano's SO arithmetic Peano's set of 6 FO axioms extended with the SO induction axiom is called *Peano's arithmetic with SO induction* axiom. We denote it $\mathtt{Th}(\mathbb{N})$. The following theorem shows that, similarly as $\mathtt{Th}(DB)$ for a database DB, $\mathtt{Th}(\mathbb{N})$ expresses all information in the structure $\mathbb{N}$.

Theorem 2.5.2. *A structure $\mathfrak{A}$ is a model of $\mathtt{Th}(\mathbb{N})$ iff $\mathfrak{A}$ is isomorphic to $\mathbb{N}$.*

It shows that $\mathtt{Th}(\mathbb{N})$ is a categorical theory.

Corollary 2.5.1. $\mathbb{N} \models \varphi$ *iff* $\mathtt{Th}(\mathbb{N}) \models \varphi$.

The FO induction schema The SO induction axiom, which formalizes $DCA(0, S)$, cannot be expressed in FO. This will be proven in Chapter 7. However, it can be approximated.

Definition 2.5.3. The *induction schema* (over Σ) is the countably infinite set of formulas of the form:

$$\forall \bar{y}((\varphi[0, \bar{y}] \wedge \forall x(\varphi[x, \bar{y}] \Rightarrow \varphi[S(x), \bar{y}])) \Rightarrow \forall x \varphi[x, \bar{y}])$$

where $\bar{y}$ is a tuple of variables and $\varphi[x, \bar{y}]$ is an arbitrary FO formula (over Σ) with free variables $x, \bar{y}$.

Each instance of the induction schema has the same structure as the induction axiom and expresses that if the set of all x satisfying $\varphi[x, \bar{y}]$ contains 0 and is closed under S, then all elements of the domain belong to it. Hence, the schema expresses the same idea as the SO induction axiom, but only for subsets of the domain that can be expressed by formulas φ. Since not all subsets can be expressed by formulas, this is strictly weaker.

The countably infinite FO theory consisting of the original 6 axioms augmented with the induction schema is called Peano arithmetic with the induction *schema*.

Exercise 2.5.9. *Use a cardinality argument to show that not all subsets of* $\mathbb{N}$ *can be expressed by formulas.*

Peano arithmetic is an approximate description of $\mathbb{N}$. All sentences of Peano arithmetic with schema are true in the natural numbers. But, this theory has non-standard models that are not isomorphic to $\mathbb{N}$ and in which domain closure is not satisfied. Nevertheless, most properties of interest in the natural numbers can be proven from Peano arithmetic with the schema.

The non-standard models differ from $\mathbb{N}$ by containing many more elements than $\mathbb{N}$. In particular, their domain consists of the natural numbers and infinitely many copies of the integer numbers.

2.5.3 Generalizing UNA and DCA

In the context of a database structure and the structure of the natural numbers, the proposition expressed as $UNA(\{0, S/1\})$ can be explained as "different terms $S^n(0)$ represent different objects". $DCA(\{0, S/1\}$ can be explained as "every object is represented by at least one term $S^n(0)$. Together, they express that all objects are represented by exactly one term.

These axioms can be generalized to express properties of constructor functions and constructed types.

Unique Name Axioms in FO(Types) Given a finite set τ of function symbols (including object symbols) with result type $\mathfrak{t}$. Then the (generalized) Unique Name Axioms for τ, denoted $UNA(\tau)$ expresses the set of functions denoted by symbols in τ are "groupwise injective": function symbols in τ denote injective functions, and the ranges of two different function symbols in τ are disjunct. This is expressed as follows.

Definition 2.5.4. $UNA(\tau)$ is the theory:

$$\forall \bar{x} \forall \bar{y} \neg (F(\bar{x}) = G(\bar{y}))$$
$$\forall \bar{x} \forall \bar{y} (F(\bar{x}) = F(\bar{y}) \Rightarrow \bar{x} = \bar{y})$$

for all function (and object symbol) symbols $F, G \in \tau$.

Special cases seen before are for $\tau = \{0, S\}$ and for τ the set of object symbols in a database structure.

In general, $UNA(\tau)$ entails that all terms consisting *only* of symbols of τ represent different objects, but it is stronger than that. It expresses that no object in the domain is more than once the result of one of the function symbols or object symbols of τ.

(Typed) Domain Closure In the context of the natural numbers, domain closure for $\tau = \{0, S/1\}$ is the proposition that every natural number is represented by at least one term $S^n(0)$. In the context of a database structure with set S of object symbols, domain closure for S is the proposition that every element of the domain is represented by at least one object symbol of S.

A natural generalization of this proposition in typed logic is that a type τ is "constructible" from the functions represented in a set τ of function and object symbols with result type $\mathfrak{t}$. In case τ does not contain recursive functions, i.e., no function symbol in τ has arguments of type $\mathfrak{t}$, this means that the domain associated with $\mathfrak{t}$ is the union of the ranges of the values of symbols in τ. In case τ contains recursive functions, it means that the domain associated with $\mathfrak{t}$ is inductively constructible by the function represented in τ.

We call this informal proposition the (generalized) domain closure for τ and it $\mathbf{DCA}(\tau)$.

Note that $\mathbf{DCA}(\tau)$ does not express the injectivity of function symbols in τ.

The domain closure $\mathbf{DCA}(\tau)$ cannot be expressed in FO, as we shall prove in Chapter 6. It can be expressed in (typed) SO or, using an auxiliary symbol, in FO(ID).

Definition 2.5.5. Let $U_\mathfrak{t}$ be an auxiliary unary predicate symbol of type $\mathfrak{t} \to \mathbb{B}$. Then $DCA^{FO(ID)}(\tau)$ is the theory consisting of

$$\forall x[\mathfrak{t}]U_\mathfrak{t}(x)$$

and the definition Δ_τ consisting of the following rule for every $F \in \tau$:

$$\forall x_1 \ldots \forall x_n(U_\mathfrak{t}(F(x_1, \ldots, x_n)) \leftarrow U_\mathfrak{t}(x_{i_1}) \wedge \ldots U_\mathfrak{t}(x_{i_m}))$$

where $i_1, \ldots, i_m$ are the argument positions of type $\mathfrak{t}$ of F.

Exercise 2.5.10. *Express* $\mathbf{DCA}(\tau_{\mathbf{list}})$ *for* $\tau_{list} = \{Nil : list, Cons : \mathfrak{t} \times list \to list\}$ *in SO and in FO(ID).*

Remark 2.5.5. The original version of the $\mathbf{DCA}(\tau)$ entails that in every model $\mathfrak{A}$ of $\mathbf{DCA}(\tau)$, for each domain element $d \in D_\mathfrak{A}$, there exists a term t over τ such that $t^\mathfrak{A} = d$. That is, each domain element is represented by a term over τ. In the extended notion of DCA, this is no longer the case. As an example, consider the structure $\mathfrak{A}$ of τ_{list} with:

- $\mathfrak{t}^\mathfrak{A} = \mathbb{N}$
- $list^\mathfrak{A}$ is the set of finite lists of natural numbers and $Nil^\mathfrak{A}$ and $Cons^\mathfrak{A}$ the corresponding functions.

It is easy to show that this structure satisfies $\mathbf{DCA}(\tau_{\mathbf{list}})$. However, in this structure, only one object is denoted by a term over τ_{list} and this is $Nil^\mathfrak{A}$, the empty list.

The combination of UNA and DCA for some vocabulary τ with result type $\mathfrak{t}$ expresses that the function symbols of τ are *constructors* and that $\mathfrak{t}$ is a type constructed from these constructors.

Remark 2.5.6. Given some vocabulary Σ, let $\mathfrak{t}$ be some type of Σ and τ the set of all function symbols with result type $\mathfrak{t}$. Then we denote $UNA(\tau)$ and $DCA(\tau)$ as $UNA(\mathfrak{t})$, respectively $DCA(\mathfrak{t})$.

Remark 2.5.7. A practical specification language should provide a convenient, flexible, compact representation of UNA and DCA. It is a weakness of FO that this is not possible.

In some declarative programming languages, including Prolog, Haskell, also database systems, UNA and DCA are assumed per default for all object and function symbols (except the defined

ones in Haskell). In the IDP language, UNA and DCA can be expressed for a subset τ of the function symbols of the type using the "is constructed from" expression.

2.6 Important for the exam

```
The theoretical part of the exam consists of small and large questions.


Small questions:  examples

- explain/correct some tricky logical formula (e.g., those from the
  pragmatics section);
- the difference between a definition and its material implications,
  or its FO predicate completion.
- evaluate a formula in a structure; show non-equivalence of two formulas by
  a structure (possible world analysis)
- explain the induction axiom in Th(NAT); what is its link with DCA? With the
  reachability problem? with inductive definitions? With HO?
- what are the three components of Th(DB), the theory of databases?
  Explain why they are needed and what they express
- how to model null values in Th(DB)
- A trick question: what is the complexity of FO?


Knowledge and understanding that I think is needed to answer the
questions  that I may ask:

Knowledge of semantics of FO and extensions of it
- values of symbols in FO and extensions of it : domains, tuples,
  relations, functions, the hierarchy of values
- structures: in graph notation, in symbolic notation, as in IDP;
- isomorphic structures

Knowledge of
- definitions of terms versus formulas
- definition of satisfaction relation
- definitions of the main semantical concepts (satisfiable,
  valid/tautological, contradictory, entailment, equivalence,
  categorical theories/complete theories) (these concepts were defined
  introduced in Chapter 1 as well)

Understanding of pragmatics (no theoretical or overview questions, but
  I may verify your understanding of FO sentences of the kind seen in
  this section and elsewhere)

Understanding of the openness problem of FO and the role of UNA, DCA
  and definitions to cope with it.
```

Understanding of extensions of FO and being able to use them (no formal definitions) : types, arithmetic, aggregates, definitions
Understanding of the SO induction axiom
Understanding of the difference between an inductive definition and sets of FO implications or FO equivalences.

Knowledge of TH(DB): the theory of a database
Understanding of the Peano axioms; knowledge of the induction axiom and the induction schema and the relation to the domain closure axiom.

Chapter 3

Modeling dynamic systems in classical logic

3.1 Introduction: Modeling dynamic systems in LTC

So, far we saw applications of logic FO(.) for modelling static applications. In this chapter, we will use it for modeling dynamic worlds, with actions and time dependent relations and functions that change over time.

Many logics to specify dynamic worlds have time built in. E.g., the logics of the next chapter: Event-B, CTL, LTL. This is not the case with FO(.). In FO(.), dynamic and temporal worlds will be modelled by making time an explicit parameter of predicates and functions. In the AI community, this is sometimes called the Method of Temporal Arguments.

This method has advantages and some disadvantages. A disadvantage is that temporal theories in FO(.) are more verbose than in temporal logics since all specific aspects of time and effect of actions need to be expressed while in temporal logics, some properties have been implemented in the logic and do not need to be expressed. On the other hand, The method of Temporal Arguments has two advantages:

- *No hidden assumptions*: there are no hidden laws; all laws of time are explicit. Therefore, this method leads to a better understanding of the foundations of temporal reasoning.
- *Flexibility*: since all aspects are explicit as axioms, they can be modified, whereas in a temporal logic, certain axioms are built in.

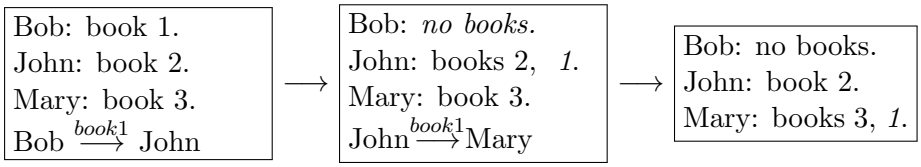
Modeling dynamic domains in AI The problem of modelling temporal domains (or dynamic domains is an important research topic in Knowledge Representation. The methodology for representing temporal domains that is presented below is based of this research. We call it the *Linear Time Calculus* (LTC).

A *Linear Time Calculus* is a typed FO(.) theory formalizing a dynamic domain. The time type `T` used in it, is interpreted by the natural numbers. This results in a time topoloy called *linear time*, since the natural numbers form a linear order. This is an order where each pair (n, m) of natural numbers is ordered: $n = m$ or $n < m$ or $m < n$. In words, for two diffferent time points, one is the future of the other or vice versa.

While this seems natural and evident, in other temporal modelling languages, time is modeled as a *tree*. This tree represents alternative potential futures; one time point or state may have multiple successor times, representing alternative ways the world may evolve. Such a time topology is called *branching time*.

Example 3.1.1. As a running example consider the following scenario. Books are owned by persons. The only action is *Give*: a person gives a book to a person. The scenario is that initially, Bob owns book 1, John owns book 2 and Mary owns book 3. First Bob gives book 1 to John. Then John gives book 1 to Mary.

Actions such as *Give* are seen as *state transformers*. This scenario results in a sequence of state transitions:



Definition 3.1.1. A *LTC vocabulary* Σ_a consists of

- A set of types, including $\mathtt{T}$ (time).
- A set of *static* predicates and function symbols, without argument of type $\mathtt{T}$.
- A set of *dynamic* predicate symbols that have exactly one argument of $\mathtt{T}$.
- Interpreted symbols of $\mathtt{T}$: numerals $0, 1, 2, \cdots : \mathtt{T}$, $S(\mathtt{T}) : \mathtt{T}$ the successor function, $\mathtt{T} + \mathtt{T} : \mathtt{T}$.

Notice that dynamic function symbols are not allowed. For simplicity, replace n-ary function symbols by a $n+1$-ary graph predicate.

Example 3.1.2. The LTC vocabulary of the running example:

- Types: $Person, Book, \mathtt{T}$
- Static: $Bob, Mary, John : Person; Book_1, Book_2, Book_3 : Book$
  - An example of a static predicate in this context: $HasAuthor(Book, Person)$.
- Dynamic:
  - $Owns(Person, Book, \mathtt{T})$: $Owns(p, b, t)$: person p owns book b at time t.
  - $Gives(Person, Book, Person, \mathtt{T})$: $Gives(g, b, r, t)$: person g (the giver) gives book b to r (the receiver) at time t.

Dynamic predicates can often be categorized as:

- *Action* predicates $Give, Drop, Move, Put, Start, \ldots$
- *Fluent* predicates which represent state Ex. Owns, Location, On, Off, Running, ...

In the LTC, we do not make an explicit distinction. However, we will see that they are often axiomatized in a distinctive way.

Some examples of expressions

- *Timed axioms.* E.g.,

$$Owns(Bob, Book_1, 0)$$
$$Gives(John, Book_1, Mary, 1)$$

- *Action preconditions* express necessary conditions for actions to happen:

$$\forall t \forall g \forall b \forall r (Gives(g, b, r, t) \Rightarrow Owns(g, b, t))$$

- *Action concurrency axioms* express what actions cannot happen simultaneously:

$$\forall t \forall g \forall b \forall r_1 \forall r_2 (Gives(g, b, r_1, t) \wedge Gives(g, b, r_2, t) \Rightarrow r_1 = r_2)$$

- *Effect axioms* express the effect of actions in successive states.

$$\forall t \forall g \forall b \forall r (Gives(g, b, r, t) \Rightarrow Owns(r, b, t + 1))$$
$$\forall t \forall g \forall b \forall r (Gives(g, b, r, t) \Rightarrow \neg Owns(g, b, t + 1))$$

Example 3.1.3. A first attempt at the book example:

$$Owns(Bob, Book_1, 0).$$
$$Owns(John, Book_2, 0).$$
$$Owns(Mary, Book_3, 0).$$
$$\forall g \forall b \forall r \forall t (Gives(g, b, r, t) \Rightarrow Owns(g, b, t))$$
$$\forall g \forall b \forall r_1 \forall r_2 \forall t (Gives(g, b, r_1, t) \wedge Gives(g, b, r_2, t) \Rightarrow r_1 = r_2)$$
$$\forall g \forall b \forall r \forall t (Gives(g, b, r, t) \Rightarrow Owns(r, b, t + 1))$$
$$\forall g \forall b \forall r \forall t (Gives(g, b, r, t) \Rightarrow \neg Owns(g, b, t + 1))$$
$$Gives(Bob, Book_1, John, 0) \wedge Gives(John, Book_1, Mary, 1)$$

Is this theory a precise description of the scenario? **Not even close.**

What is missing?

- Missing UNA, DCA for $Person, Book$.

  Exercise 3.1.1. *Think of possible worlds in which UNA, DCA are not satisfied and how they differ from the intended world. Think of propositions that we expect to be true in the intended world but are false in these other possible worlds and hence, are not entailed by this theory. E.g., what if $Bob = John$? What if there is another book?*

- No *definitions* of initial state and actions.
  - We did not express that initially, Bob only owns book 1, John only book 2, ... Hence this theory does not entail that $\neg Owns(Bob, Book_3, 0)$ nor $\neg \exists b Owns(Bob, b, 2)$.
  - Likewise, we did not express that there are only two events. Hence this theory does not entail that $\neg Gives(Mary, Book_3, John, 1)$, and therefore, also not that $Owns(Mary, Book_3, 2)$.

- We did not represent that the two represented effects of *Give* are its only effects. E.g., if $Gives(Bob, Book_1, John, 0)$ is the only action at time 0, then all *Owns* atoms except for $Owns(Bob, Book_1, \dots)$ and $Owns(John, Book_1, \dots)$ have the same value at time 1 and 2. Hence this theory does not entail that $Owns(John, Book_2, t)$ at $t = 1, 2$.

  We need to add *inertia* axioms, to express what is not affected by the actions in successive states.

$$\begin{aligned}
&\forall t \forall g \forall b \forall r \forall p \forall b1 (Owns(p, b1, t) \wedge Gives(g, b, r, t) \wedge \neg(g = p \wedge b = b1) \\
&\qquad \Rightarrow Owns(p, b1, t + 1)) \\
&\forall t \forall g \forall b \forall r \forall p \forall b1 (\neg Owns(p, b1, t) \wedge Gives(g, b, r, t) \wedge \neg(g = p \wedge b = b1) \\
&\qquad \Rightarrow \neg Owns(p, b1, t + 1))
\end{aligned}$$

  What if there are multiple fluents and actions? For every combination of fluent and action, an axiom of this kind is required. This is complex. We need to find a better way.

Exercise 3.1.2. *Apply the IDP system on the given (incomplete) theory to verify the above claims about its incompleteness.*

`http://dtai.cs.kuleuven.be/krr/idp-ide/?present=Book1`

3.2 Intermezzo: Some history of temporal reasoning in AI

Origins of logic-based AI

- Around 56-57, the first ideas on logic-based Artificial Intelligence arose, in the context of temporal reasoning in FO under the leadership of John McCarthy.

- McCarthy 57: the historically first FO theory of actions and their effects on properties: *Situation Calculus*

John McCarthy (1927-2011)

- Coined the term *artificial intelligence* (1957)
- Developed the first approach to formal modelling of dynamic systems *Situation calculus* (57)
- Developed the first functional language Lisp (60)
- Created the Stanford AI lab, educating many good AI researchers.
- Published with Hayes (69) a famous paper on the frame problem in FO. This way, he became founding father of the field of Nonmonotonic Reasoning, a subfield of Knowledge Representation. He remained active in this field till his death in 2011.
- One of the first to prove the correctness of a compiler (70).
- Turing Award 1971.
- Remained influential in the 80ties and 90ties.

A historical note : The frame problem In a paper in 57, in the early days of Artificial Intelligence, McCarthy and co-authors had presented an initial version of the situation calculus in FO. They had observed that expressing effects of actions was fairly easy. In a seminal paper in 1969, McCarthy&Hayes pointed to several problems with this early situation calculus. It is fairly easy to represent the effects of actions in FO (as we saw in the book example). However, this does not suffice. Also non-effects must be described. I.e., we need to describe what properties remain unchanged when some event happens. Such axioms will be called *inertia axioms*.

The problem is that in a complex world, every action affects only a very small set of properties, leading to a small number of effect axioms, but leaves a myriad of properties unchanged, leading to a very large number of inertia axioms. E.g., in the case of the *Give* action:

- $\forall \ldots Owns(p_1, b_1, s) \wedge (p_1 \neq p \wedge b_1 \neq b) \Rightarrow Owns(p_1, b_1, Do(Gives(p, b, r), s))$
- $\forall \ldots TV(On, s) \Rightarrow TV(On, Do(Gives(p, b, r), s))$
- $\forall \ldots TV(Off, s) \Rightarrow TV(Off, Do(Gives(p, b, r), s))$
- $\forall \ldots On(Pen, Table, s) \Rightarrow On(Pen, Table, Do(Gives(p, b, r), s))$

This is clearly unfeasible. The problem of expressing the inertia axioms is part of the *frame problem* that was discussed by McCarthy and Hayes (69).

Definition 3.2.1. The frame problem is the problem to express effects and non-effects of actions and events.

(Part of) the problem they saw was that explicit inertia axioms are required for every combination of action and fluents. When modelling a complex world, there will be too many combinations

of actions and fluents. This also leads to a lack of *elaboration tolerance*. When building a theory of a complex world, it is to be expected that the theory is to be updated many times with new fluents or actions. Each subsequent update seems to require an increasing number of inertia rules.

Definition 3.2.2. (from John McCarthy (1998)) A formalism is elaboration tolerant to the extent that it is convenient to modify a set of facts expressed in the formalism to take into account new phenomena or changed circumstances.

In an elaboration tolerant logic, it should be feasible to add new information to a theory in an incremental, well-structured way, with few, well-localized modifications to the existing theory and/or with new axioms that correspond to the information. Adding explicit are specific for the information.

The problem of describing inertia rules is only a small part of the full frame problem. The full frame problem is the problem of building complete descriptions of the everyday world, to obtain the sort of theory that early AI researchers thought to use to build AI systems. The complexity of the real world is endless: the number of properties, actions, agents; the number of unexpected events and behaviours, mutual influences, ramifications, etc. There is simply too much complexity to build a complete description of it. The full frame problem remains unsolved, and is considered to be unsolvable by many.

The complexity of systems currently under consideration in formal modelling paradigms is *insignificant* compared to that of the everyday world. Nevertheless, there is an abundance of "small" systems for which it would be useful to formally model them. Even for such "small" systems, the inertia problem observed by McCarthy and Hayes was serious and needed to be solved for formal modelling to become feasible.

Between 69 till the late eighties, it was thought to be impossible to express inertia in a concise and elaboration tolerant FO theory.

It lasted until around 1990 before a correct, elegant and general solution for the frame problem for situation calculus in FO was found. This was mainly the work of Ray Reiter.

> *Ray Reiter, Knowledge in Action: Logical Foundations for Specifying and Implementing Dynamical Systems, MIT Press, 2001.*

The method presented in this course is inspired by Reiters method, but differs as follows:

- We use a Linear Time Calculus instead of Situation Calculus.
- We use FO(ID) definitions to express inertia and causal rules rather than FO. However, the completion of these definitions correspond to Reiters solution.

Remark 3.2.1. In the early days of artificial intelligence, there was great focus on "grand scale" knowledge representation: representing relevant knowledge of a substantial area of human expertise, providing the knowledge base for an artificial agent to act intelligently in the real world. Building such a knowledge base of logic expressions is an immense problem that is still unsolved. Many AI researchers believe it cannot be done. Since then, knowledge representation turned to smaller scale problems. The solution provided by Reiter works fine for specific temporal domains, but it does not address the original ambitions of grand scale KR.

3.3 Expressing Inertia through causal laws

Definition 3.3.1. *Inertia* is the property that a fluent's value does not change in time unless it is explicitly affected by an action.

We distinguish between two types of dynamic symbols:

- *Inertial* symbols: its values persist unless directly affected by an action. Fluent symbols typically have inertia. Ex. Owns.
- *Non-inertial* symbols: its values tend not to persist in a next state. Action predicates typically do not have inertia. Ex. Give.

Most fluents are inertial. Most actions have relatively few effects which are easy to understand.

It is relatively easy to represent what are the effects of actions. E.g.,

$$\forall t \forall g \forall b \forall r(Gives(g,b,r,t) \Rightarrow Owns(r,b,t+1))$$
$$\forall t \forall g \forall b \forall r(Gives(g,b,r,t) \Rightarrow \neg Owns(g,b,t+1))$$

How to represent the inertia of inertial fluents under other actions? We hope for compact representations, in which we can focus on the effects, and get the inertia for free.

Causality When we express:

$$\forall t \forall g \forall b \forall r(Gives(g,b,r,t) \Rightarrow Owns(r,b,t+1))$$
$$\forall t \forall g \forall b \forall r(Gives(g,b,r,t) \Rightarrow \neg Owns(g,b,t+1))$$

what we really want to say is this:

$$\forall t \forall g \forall b \forall r(Gives(g,b,r,t)\text{" }\textbf{causes}\text{ "}Owns(r,b))$$
$$\forall t \forall g \forall b \forall r(Gives(g,b,r,t)\text{" }\textbf{causes}\text{ "}\neg Owns(g,b))$$

In natural language, we frequently use conditionals to express that something causes something else. However, to say that x *causes* y is not the same as the material implication $x \Rightarrow y$! Causal conditionals are yet another type of conditional. The question is how we can express x *causes* y in FO(.)?

The key is to specify exactly what the effects of actions are, that is what the changes are that they bring about: what fluents they cause to become true and what fluents they cause to become false. We can then define the fluents at every state point in terms of these effects in the previous state.

A solution for the frame problem To specify what causal effects take place, we introduce for each fluent P two new auxiliary *causality* predicates, to express causes to make P true, and causes to make P false. We specify the existing effects in a precise way by *defining* these auxiliary symbols. Then we specify the inertial fluents by *defining* its value in a state by defining it in terms of its previous state and the effects that happened in the previous state.

Per n-ary inertial predicate $P(\bar{x}, t)$, we introduce three new predicates with the following intended interpretation:

- $I\_P(\bar{x})$: Initially, at $t = 0$, $P(\bar{x}, t)$ is true.
- $CT\_P(\bar{x}, t)$: At t, $P(\bar{x}, t)$ is Caused to become True
- $CF\_P(\bar{x}, t)$: At t, $P(\bar{x}, t)$ is Caused to become False.

The core of an LTC is one large FO($\cdot$) definition Δ defining every inertial fluent P/n and its causality predicates $CT\_P/n, CF\_P/n$ by simultaneous induction on the standard order on time points. The rules are of the following kind:

$$\begin{array}{l}\forall \bar{x}(P(\bar{x}, 0) \leftarrow I\_P(\bar{x})) \\ \forall \bar{x} \forall t(P(\bar{x}, t+1) \leftarrow CT\_P(\bar{x}, t)) \\ \forall \bar{x} \forall t(P(\bar{x}, t+1) \leftarrow P(\bar{x}, t) \wedge \neg CF\_P(\bar{x}, t)) \\ \ldots \text{Effect rules as definitional rules for } CT\_P, CF\_P \ldots\end{array}$$

The resulting definition Δ defines multiple predicates by simultaneous induction. It is a definition over an induction order, being the standard order of natural numbers on `T`. The definition is nonmonotone due to the negative occurrence $\neg CF\_P(\bar{x}, t)$ of a defined predicate in the third rule. In case $CF\_P(\bar{x}, t)$ is defined in terms P, P is inductively defined in terms of itself via a nonmonotone inductive rule, just like $\mathfrak{A} \models \neg\varphi$ is defined nonmonotonically in terms of $\mathfrak{A} \models \varphi$.

The first 3 sorts of rules are called the *frame rules*:

- the base case defines P in the initial state;
- the 2nd inductive rule defines P to be true if there is a cause for it in the previous state; it is an inductive rule since CT_P is a defined predicate of Δ;
- the 3rd inductive rule defines that P remains true unless it is caused false in the previous state. This rule is called the *inertia rule* or *inertia law*. It is a nonmonotone rule.

Effect rules are definitional implications of causality predicates and specify exactly what effects do occur; no other effects occur.

The recursion underlying this definition can be shallow or deep. I call it shallow if all effect rules are non-recursive. In that case, the only recursion is due to the inertia rule. E.g., the definition Δ for the running example would be shallow, since there is only one action $Give$ and its effect rules are non-inductive:

$$\begin{array}{l}\forall g \forall b \forall r \forall t(CT\_Own(r, b, t) \leftarrow Gives(g, b, r, t)) \\ \forall g \forall b \forall r \forall t(CF\_Owns(g, b, t) \leftarrow Gives(g, b, r, t))\end{array}$$

We now formally define the expressions in a LTC.

Rules and axioms of the Linear Time Calculus Given is an LTC vocabulary Σ_a.

Definition 3.3.2. A formula or rule φ over Σ_a is a *(single) state* formula or rule in variable t if t has only free occurrences and the only term of sort `T` in φ is t.

A state formula is a proposition describing the state at time t.

- E.g., $\forall g \forall b \forall r(Gives(g, b, r, t) \Rightarrow Owns(g, b, t))$.

Definition 3.3.3. A formula or rule φ over Σ_a is a *bi-state* formula or rule in variable t if t has only free occurrences and the only terms of sort `T` in φ are $t, t+1$.

A bi-state formula is a proposition connecting the states at two successive times $t, t+1$.

- E.g., $\forall g \forall b \forall r(Gives(g, b, r, t) \Rightarrow \neg Owns(g, b, t+1))$

A *single state formula* is a bi-state formula in which $t+1$ do not occur.

Sometimes, $S(t)$ or $Next(t)$ is used to denote $t+1$.

Definition 3.3.4. An *action effect rule* is a definitional implication of the form:

$$\forall t \forall \bar{x}(CT\_P(\bar{a}, t) \leftarrow Act(\bar{a}', t) \wedge \varphi[t])$$
$$\forall t \forall \bar{x}(CF\_P(\bar{a}, t) \leftarrow Act(\bar{a}', t) \wedge \varphi[t])$$

where Act is an action predicate symbol, $P/n+1$ inertial, $\bar{a}, \bar{a}'$ tuples of terms of suitable length and type, $\varphi[t]$ a single state formula in t.

We call an action effect rule *context independent* if $\varphi[t] = true$. Otherwise, it is *context dependent*.

Context independent effect rules are:
- $\forall t \forall g \forall b \forall r(CT\_Owns(r, b, t) \leftarrow Gives(g, b, r, t))$
- $\forall t \forall g \forall b \forall r(CF\_Owns(g, b, t) \leftarrow Gives(g, b, r, t))$

Context dependent:
- $\forall t \forall x(CT\_Broken(x, t) \leftarrow Drop(x, t) \wedge Fragile(x))$
- E.g., picking up a block from a stack of blocks makes the underlying block clear: $\forall t \forall b \forall b1(CT\_Clear(b, t) \leftarrow Pick(b1, t) \wedge On(b1, b, t))$

Definition 3.3.5. An *action precondition axiom* for action predicate Act is of the form

$$\forall \bar{x} \forall t(Act(\bar{x}, t) \Rightarrow \varphi[t])$$

where $\varphi[t]$ is a state formula.

It expresses a necessary condition for the action to occur. E.g., to give a book away, you need to own it.

$$\forall t \forall g \forall b \forall r (Gives(g, b, r, t) \Rightarrow Owns(g, b, t))$$

Definition 3.3.6. The *no-concurrency axiom* for action predicates $Act_1, \ldots, Act_n$ expresses that at most one action takes place at the same time.

To express that no actions of different types occur simultaneously, but also that at most one instance of each type of action occurs simultaneously.

In FO, if there are n action predicates, $n + \frac{n \times n-1}{2}$ axioms are required: one per action predicate, one per pair of different action predicates.

In FO(.), it could be expressed by a single axiom of length n:

$$\forall t(\#\{\bar{x}_1 : Act_1(\bar{x}_1, t)\} + \cdots + \#\{x_n : Act_n(\bar{x}_n, t)\} \leq 1)$$

E.g., in the running example:

$$\forall t(\#\{(g, b, r) : Gives(g, b, r, t)\} \leq 1)$$

or

$$\forall g \forall b \forall r \forall t(Gives(g, b, r, t) \wedge Gives(g1, b1, r1, t) \Rightarrow g = g1 \wedge r = r1 \wedge b = b1)$$

Definition 3.3.7. A base *Linear Time Calculus* over Σ_a is a theory consisting of an LTC definition Δ_a consisting of *frame rules* and *action effect rules* for every inertial fluent, of *action preconditions axioms*, the *no-concurrency axiom* and *initial state expressions* and static expressions (definitions and/or sentences).

Example 3.3.1. Expressing the running example in LTC

$$\begin{array}{l}
UNA(Person) \wedge DCA(Person) \\
UNA(Book) \wedge DCA(Book) \\
\left\{ \begin{array}{l} I_Owns(Bob, B1) \leftarrow \\ I_Owns(John, B2) \leftarrow \\ I_Owns(Mary, B3) \leftarrow \end{array} \right\} \\
\left\{ \begin{array}{l} Gives(Bob, B1, John, 0) \leftarrow \\ Gives(John, B1, Mary, 1) \leftarrow \end{array} \right\} \\
\left\{ \begin{array}{l} \forall p \forall b(Owns(p, b, 0) \leftarrow I_Owns(p, b)) \\ \forall p \forall b \forall t(Owns(p, b, t+1) \leftarrow CT_Owns(p, b, t)) \\ \forall p \forall b \forall t(Owns(p, b, t+1) \leftarrow Owns(p, b, t) \wedge \\ \qquad\qquad \neg CF_Owns(p, b, t)) \\ \forall g \forall b \forall r \forall t(CT_Owns(r, b, t) \leftarrow Gives(g, b, r, t)) \\ \forall g \forall b \forall r \forall t(CF_Owns(g, b, t) \leftarrow Gives(g, b, r, t)) \end{array} \right\} \\
\forall g \forall b \forall r \forall t(Gives(g, b, r, t) \Rightarrow Owns(p, b, t)) \\
\forall t(\#\{(g, b, r) : Gives(g, b, r, t)\} \leq 1)
\end{array}$$

Representation LTC in IDP-language

`http://dtai.cs.kuleuven.be/krr/idp-ide/?present=Book2`

- It uses the type name `Time` to represent the time type `T`.
- IDP can only handle finite domains, hence $Time$ should be restricted to finite type:
  - In the declaration of the vocabulary, we insert `type Time isa Int` to obtain arithmetical symbols for type `Time`.
  - In the declaration of the IDP structure, declare `Time = {0..N}`, with `N` some numeral.

One needs to pay attention with universal quantification over `Time` in formulas of the IDP language! Suppose `P(Time)` is declared. Consider the formula

$$\texttt{P(0)} \land \forall \texttt{t[Time]} : \texttt{P(t)} \Rightarrow \texttt{P(t + 1)}.$$

This sentence is unsatisfiable in IDP since it entails `P(N+1)` which is a contradiction with P's type declaration. Indeed, the latter entails that P is a subset of `Time` which does not contain `N+1`.

Such errors occur in quantified formulas over a finite integer type, where the quantified variable `t` also occurs in atoms `P(t+1)`. This may lead to **inconsistency** or **the unintended disappearance** of models. They may be difficult to discover.

The correction is:

$$\texttt{P(0)} \land \forall \texttt{t[Time]} : \texttt{t} < \texttt{MAX[: Time]} \land \texttt{P(t)} \Rightarrow \texttt{P(t + 1)}.$$

The extra condition takes care that `t` is quantified over `[0,N-1]`.

The problem does not occur in definitions. E.g.,

$$\left\{ \begin{array}{l} P(0) \leftarrow . \\ \forall n : P(n+1) \leftarrow P(n). \end{array} \right\}$$

The same problem seemingly appears in the inductive rule. However, IDP internally interprets this rule as follows:

$$\left\{ \begin{array}{l} P(0) \leftarrow . \\ \forall m[Time] : P(m) \leftarrow \exists n[Time] : m = n + 1 \land P(n). \end{array} \right\}$$

Now you can see that this rule cannot derive `P(N+1)`.

Later, we will often use `Start` for 0 and `Next(Time):Time` for $S/1$. Some inference procedures of IDP require this (progression inference, proof of invariance).

3.3.1 Discussion of LTC

Assymmetries in LTC An LTC contains causation and inertial rules for P:

$$\forall \bar{x} \forall t(P(\bar{x}, t+1) \leftarrow CT\_P(\bar{x}, t))$$
$$\forall \bar{x} \forall t(P(\bar{x}, t+1) \leftarrow P(\bar{x}, t) \land \neg CF\_P(\bar{x}, t))$$

Strikingly, there are no symmetrical causation and inertial rules for $\neg P$ of the following kind:

$$\forall \bar{x} \forall t(\neg P(\bar{x}, t+1) \leftarrow CF\_P(\bar{x}, t))$$
$$\forall \bar{x} \forall t(\neg P(\bar{x}, t+1) \leftarrow \neg P(\bar{x}, t) \wedge \neg CT\_P(\bar{x}, t))$$

Such rules could never be present in a theory, since such rules are **not legal FO(ID) syntax.** Indeed, rules with negated head are not allowed in definitions. [1]

However, let us consider the corresponding material implications:

$$\forall \bar{x} \forall t(CF\_P(\bar{x}, t) \Rightarrow \neg P(\bar{x}, t+1))$$
$$\forall \bar{x} \forall t(\neg P(\bar{x}, t) \wedge \neg CT\_P(\bar{x}, t) \Rightarrow \neg P(\bar{x}, t+1))$$

The first expresses causation of $\neg P$, the second inertia for $\neg P$. The question is then whether these axioms are implied by the frame rules of LTC. Well, it can be proven easily that the second implication, expressing inertia for $\neg P$ is indeed entailed by the fluent definitions with its frame rules: a fact that is false at t, remains false at $t+1$ unless there is a cause to make P true.

Exercise 3.3.1. *Explain this intuitively.*

However, the first implication which states that a cause for $\neg P$ leads to $\neg P$ at $t+1$ does not always hold. Nevertheless, under "reasonable" condition to be explained below, also this property is entailed by the frame definition of P.

Assume that an LTC theory allows for contradictory effects, that is, it is possible that at some time t, both $CT\_P(t)$ and $CF\_P(t)$ holds, expressing that there is a cause for P to be become true, but also to become false. What happens then? To investigate this, we need to look at the frame definitional rules. We may observe that in this case the second frame rule unambiguously expresses that P is true at $t+1$. Hence, P is true at $t+1$, even if there is a cause to be false. The cause for P dominates the cause for $\neg P$.

It follows that the "reasonable conditions" under which an cause for $\neg P$ leads to $\neg P$ at $t+1$ is that effect rules do not specify contradictory effects. More precisely, it can be shown that if an LTC entails

$$\forall t(\neg(CT\_P(\bar{x}, t) \wedge CF\_P(\bar{x}, t))$$

for a fluent P, then it entails

$$\forall \bar{x} \forall t(CF\_P(\bar{x}, t) \Rightarrow \neg P(\bar{x}, t+1))$$

Example 3.3.2. Surprisingly, the book LTC does not exclude contradictory effects. It does not entail

$$\forall t \forall p \forall b(\neg(CT\_Owns(x, b, t) \wedge CF\_Owns(x, b, t))$$

Exercise 3.3.2. *Find a situation in which Owns is caused to be true and false simultaneously. For help, you may go to the Book theory on*

http://dtai.cs.kuleuven.be/krr/idp-ide/?present=Book4

At the bottom, it contains the proposition that an Owns fact is both caused to be true and false simultaneously. Compute a model of this theory and discover a situation where the same Owns fact is caused to be true and false simultaneously.

[1] A frequent error in exams is to include definitional rules with negation in the head in a LTC theory. That is a basic mistake against the syntax of definitions in FO(.).

Contradictory Causation What happens when contradictory effects occur at some time t. In such a case atoms $CF\_P(\bar{x},t), CT\_P(\bar{x},t)$ are true simultaneously. It is easy to verify in the frame rules, that in such a case, the positive effect dominates the negative effect. Hence, $P(\bar{x},t)$ is derived due to the second rule of the frame definition.

It makes sense to request from the user that his LTC excludes contradictory causation. It could be verification task for an LTC theory to prove the following, for every inertial predicate:

$$\forall t(\neg(CT\_P(\bar{x},t) \wedge CF\_P(\bar{x},t))$$

Exercise 3.3.3. *Refine the Book-LTC so that it entails the above proposition.*

Exercise 3.3.4. *Show that a theory allowing for contradictory causation does not entail the following material implications.*

$$\forall \bar{x} \forall t(CF\_P(\bar{x},t) \Rightarrow \neg P(\bar{x},t+1))$$

Does it entail both positive and negative inertial laws?

$$\forall \bar{x} \forall t(P(\bar{x},t) \wedge \neg CF\_P(\bar{x},t) \Rightarrow P(\bar{x},t+1))$$
$$\forall \bar{x} \forall t(\neg P(\bar{x},t) \wedge \neg CT\_P(\bar{x},t) \Rightarrow \neg P(\bar{x},t+1))$$

Simplifying the translation For simple examples, it may make sense to drop the auxiliary symbols I_P, CT_P, CF_P and substitute their defining expressions for them.

Example 3.3.3. The running example simplified: T^c_{Book}:

$$\begin{array}{l}
UNA(Person) \wedge DCA(Person) \\
UNA(Book) \wedge DCA(Book) \\
\left\{ \begin{array}{l} Gives(Bob, B1, John, 0) \leftarrow \\ Gives(John, B2, Mary, 1) \leftarrow \end{array} \right\} \\
\left\{ \begin{array}{l} Owns(Bob, B1, 0) \leftarrow . \\ Owns(John, B2, 0) \leftarrow . \\ Owns(Mary, B3, 0) \leftarrow . \\ \forall g \forall b \forall r \forall t(Owns(r, b, t+1) \leftarrow Gives(g, b, r, t)) \\ \forall g \forall b \forall r \forall t(Owns(p, b, t+1) \leftarrow Own(p, b, t) \wedge \\ \qquad\qquad \neg \exists r(Gives(p, b, r, t))) \end{array} \right\} \\
\forall g \forall b \forall r \forall t(Gives(g, b, r, t) \Rightarrow Owns(g, b, t)) \\
\forall t \forall g \forall b \forall r_1 \forall r_2(Gives(g, b, r_1, t) \wedge Gives(g, b, r_2, t) \Rightarrow r_1 = r_2)
\end{array}$$

Eliminating auxiliary predicates is very simple in this example because the book example contains only one action and one fluent.

For more complex theories with multiple fluents and actions and effects, eliminating I_P, CT_P, CF_P leads to messy theories at best, and errors at worst.

In the project and the exam, it is not allowed to eliminate the auxiliary predicates unless explicitly stated.

From STRIPS to LTC STRIPS:

- Stanford Research Institute Problem Solver

- An automated planner developed by Richard Fikes and Nils Nilsson in 1971 at SRI International.
- Name of the language used by them to describe planning domains.
- This language is the basis of most current planning languages
- Introduction by example, and translation to LTC

We illustrate the language with an example.

Example 3.3.4. A monkey is at location A in a lab. There is a box in location C. The monkey wants the bananas that are hanging from the ceiling in location B, but it needs to move the box and climb onto it in order to reach them.

```
Initial state:  At(a), Level(low), BoxAt(c), BananasAt(b)
Goal state:  Have(Bananas)
Actions:
Move(X, Y) //move from X to Y
Preconditions:  At(X), Level(low)
Postconditions:  not At(X), At(Y)
ClimbUp(Loc) //climb up on the box
Preconditions:  At(Loc), BoxAt(Loc), Level(low)
Postconditions:  Level(high), not Level(low)
ClimbDown(Loc) //climb down from the box
Preconditions:  At(Loc), BoxAt(Loc), Level(high)
Postconditions:  Level(low), not Level(high)
MoveBox(X, Y) //monkey moves box from X to Y
Preconditions:  At(X), BoxAt(X), Level(low)
Postconditions:  BoxAt(Y), not BoxAt(X), At(Y), not At(X)
TakeBananas(Loc) //take the bananas
Preconditions:  At(Loc), BananasAt(Loc), Level(high)
Postcondition:  Have(bananas)
```

Translation STRIPS → LTC Special aspects of STRIPS:

- There is no concurrency.
- $UNA + DCA$ for all types.
- Every dynamic symbol is either inertial or an action.
- No context dependent action preconditions

Translation STRIPS → LTC in IDP is easy.
`http://dtai.cs.kuleuven.be/krr/idp-ide/?present=Monkey`

Historical note STRIPS was the first special purpose planning language built in 1991. Later languages built on top of that. Currently, the language PDDL ("Planning Domain Definition Language") is used in the AAAI planning competitions. There are many variants and extensions of PDDL. LTC generalizes most of them.

3.3.2 Generalizations of the LTC

Now we can extend LTC in various ways to accommodate for special features of the dynamic system.

Delayed effects A delayed effect rule:
$\forall p \forall t(C\_Receive(p, t+5) \leftarrow Send(p,t))$
It takes 6 days for a message p to be received.

Ramifications Sometimes, effects cause other effects: they propagate. We call such derived effects *ramifications*. Such ramifications can often be expressed elegantly, by causal rules with causality predicates in the body.

Example 3.3.5. E.g., a suitcase has a spring opening mechanism and two locks. The action of opening one lock has the ramification that it opens the suitcase if the other lock is open already. We can write rules like:

$$\begin{aligned}
&\forall t(CT_OpenSuitCase(t) \leftarrow \\
&\qquad CT_OpenLock1(t) \wedge OpenLock2(t)) \\
&\forall t(CT_OpenSuitCase(t) \leftarrow \\
&\qquad CT_OpenLock2(t) \wedge OpenLock1(t)) \\
&\forall t(CT_OpenSuitCase(t) \leftarrow \\
&\qquad CT_OpenLock2(t) \wedge CT_OpenLock1(t))
\end{aligned}$$

Note that ramifications expressed like this occur simultaneously with the effects that cause them.

A simpler representation is by a ramification rule:

$$\begin{aligned}
&\forall t(CT_OpenSuitCase(t) \leftarrow \\
&\qquad OpenLock1(t+1) \wedge OpenLock2(t+1))
\end{aligned}$$

Any set of actions leading to two locks being open will cause the suitcase being open simultaneously.

Cyclic ramifications Ramification may lead to cycles.

Example 3.3.6. Take a set of interconnected gearwheels. An action $Rotate(Gearwheel, \mathtt{T})$ causes the gearwheel to turn. All connected gearwheel are then caused to turn immediately.

Using a predicate $Conn/2$ to represent the symmetric graph of connected gearwheels, we represent this:

$$\left\{ \begin{array}{l}
\forall g \forall t(Turn(g,0) \leftarrow I_Turn(g)). \\
\forall g \forall t(Turn(g,t+1) \leftarrow CT_Turn(g,t)). \\
\forall g \forall t(Turn(g,t+1) \leftarrow Turn(g,t) \wedge \neg CF_Turn(g,t)). \\
\forall g \forall t(CT_Turn(g,t) \leftarrow Rotate(g,t)) \\
\forall g \forall g1 \forall t(CT_Turn(g,t) \leftarrow CT_Turn(g1,t) \wedge Conn(g,g1))
\end{array} \right\}$$

Notice that the last is a reachability rule. This LTC cannot be expressed in FO.

See `http://dtai.cs.kuleuven.be/krr/idp-ide/?present=Gear.simpler`
This IDP-file illustrate cyclic ramifications and also that the weaker theory obtained by applying predicate completion to the above definition is too weak and has unintended models.

Prevoyant action preconditions axioms In rare cases, it might be useful to state an action precondition in terms of the future state. E.g., a chess player pl is not allowed to move a piece p to position pos if pl would be in check as a result:

$$\forall t \forall pl \forall p \forall pos(ChessMove(pl, p, pos, t) \Rightarrow \neg Check(pl, t+1)))$$

Expressing the no-concurrency axiom in the IDP-language The no-concurrency axiom forbids more than one action per time point.

As we saw, it is tedious to express the no-concurrency axiom in FO if there are multiple actions symbols. It is easier to express with aggregates:

$$\forall t : \#\{x : Action_1(x,t)\} + \cdots + \#\{x : Action_n(x,t)\} \leq 1$$

It can be further simplified by reifying the action predicates, i.e., by introducing a type *Action*, constructors for all actions without the time argument, and a predicate $Happens(Action, \mathtt{T})$ to express that some action occurs.

Example 3.3.7. We declare

```
type Action constructed by {Gives(Person,Book,Person):Action }
```

All atoms $Gives(g, b, r, t)$ are replaced by $Happens(Gives(g, b, r), t)$. No-concurrence can be expressed compactly:

$$\forall t : \#\{a : Happens(a, t)\} \leq 1$$

Relaxing the no-concurrency axiom The no-concurrency axiom forbids more than one action and is sometimes too strict. However, typically not all actions can occur simultaneously. Therefore, *action concurrency axioms* are necessary.

E.g., an axiom to express that simultaneous *Gives* actions are allowed, but not of the same book.

$$\forall t \forall g \forall b \forall r_1 \forall r_2(Gives(g, b, r_1, t) \wedge Gives(g, b, r_2, t) \Rightarrow r_1 = r_2)$$

Defined fluents A *fluent definition* is a definition of a **non-inertial** predicate symbol with definitional rules of the form:

$$\forall \bar{x} \forall t(P(\bar{a}, t) \leftarrow \varphi[t])$$

where $\varphi[t]$ is a state formula.

- E.g., a definition of a book owner: $\{\ \forall p \forall t(Owner(p,t) \leftarrow \exists b Owns(p,b,t))\ \}$
- E.g., an object is clear if nothing is on top of it: $\{\ \forall b \forall t(Clear(b,t) \leftarrow \neg \exists b1 On(b1,b,t))\ \}$

- E.g., a player is in check if his king is attacked by a piece of the opponent:

$$\{\ \forall t \forall pl(Check(pl,t) \leftarrow\ \exists p\ Attacks(p, King(pl), t)))\ \ \}$$

A defined fluent evolves with time, due to changes on its parameters. Often it is possible to express a defined fluent as a standard fluent symbol by frame rules together with its effect rules. E.g., the effect rules for the set of book owners are:

$$\forall p \forall t(CT\_Owner(p,t) \leftarrow \exists g \exists b\ Gives(g,b,p,t))$$
$$\forall p \forall t(CF\_Owner(p,t) \leftarrow \exists r \exists b\ (Gives(p,b,r,t) \wedge \forall b1(Owns(p,b1,t) \Rightarrow b1 = b)))$$

However, it is often much simpler to express the definition than to express these effects rules.

Nondeterministic causation Some actions have nondeterministic effects.

- E.g., A lottery.
- E.g., Throwing a die results in 1 of 6 possible faces.
- E.g., When a sender transmits a message, the receiver may receive it, or the signal may be lost.

Problem: the definition of the causality predicates imposes *deterministic* effect in terms of action predicates. This does not work for nondeterministic effects.

The challenge is to have a *modular* solution: one that correctly expresses the (deterministic or non-deterministic) effect of all actions by separate causal rules.

To this end, we introduce undefined action specific causality predicates CT_P_Act to be able to express locally under which conditions these effects may take place. A solution:

- Introduce for each combination of action A and non-deterministic effect on fluent P its own causality predicate CT_P_A (or CF_P_A).
- Add rule $\forall \bar{x} \forall t(CT\_P(\bar{x},t) \leftarrow CT\_P\_A(\bar{x},t))$.
- Now, express the non-deterministic effect of A on P by some FO axiom on $CT\_P\_A(\bar{x},t)$.

This solution preserves the structure of LTC and yields a modular and elaboration tolerant solution.

Example 3.3.8. Russian Roulette

Reconsider the effect of shooting on being alive:

$$\left\{\begin{array}{l} \ldots (2 \text{ frame rules for Alive; } 1 \text{ effect rule}) \ldots \\ \forall t(Alive(t+1) \leftarrow CT\_Alive(t)) \\ \forall t(Alive(t+1) \leftarrow Alive(t) \wedge \neg CF\_Alive(t)) \\ \forall t(CF\_Alive(t) \leftarrow Shoot(t)) \end{array}\right\}$$

Now, we want to add a shooting action $RR(t)$ of Russian Roulette. It may or may not kill. We introduce and formalize $CF\_Alive\_RR(t)$. We add one definitional rule to above definition:

$$\forall t(CF\_Alive(t) \leftarrow CF\_Alive\_RR(t))$$

We express what the effect of Russian Roulette is:

$$\forall t(CF\_Alive\_RR(t) \Rightarrow RR(t))$$

The effect of getting killed by Russian Roulette is non-deterministic. It occurs only in case of a Russian Roulette shooting, but it will not necessarily occur in that case.

Example 3.3.9. Lottery in the running example

- Action $Lottery(Book, \mathtt{T})$ assigns a book (without owner) to an arbitrary person.
- Action specific Causality predicate $CT\_Owns\_Lott(p, b, t)$: there is a cause for person p to win b at time t with the lottery.

$$\left\{ \begin{array}{l} \ldots \\ \forall p \forall b \forall t(CT\_Owns(p, b, t)) \leftarrow CT\_Owns\_Lott(p, b, t) \\ \ldots \end{array} \right\}$$

$\forall p \forall b \forall t(CT\_Owns\_Lott(p, b, t) \Rightarrow Lottery(b, t))$
$\forall b \forall t(Lottery(b, t) \Rightarrow \#\{p : CT\_Owns\_Lott(p, b, t)\} = 1)$

Exercise 3.3.5. *Do a possible world analysis to show that the following sentence is not equivalent to the solution in the previous example and admits unintended models.*

$$\forall p \forall b \forall t(Lottery(b, t) \Leftrightarrow \#\{p : CT\_Owns\_Lott(p, b, t)\} = 1)$$

Hint: find a model in which there are 2 or more persons at some time t satisfying CT_Owns_Lott but there is no lottery at t.

Exercise 3.3.6. *Insert the lottery action in the IDP formalization of the book-theory and compute a model.*

Exercise 3.3.7. *A die can be picked up and can be thrown. A die on the ground has a face which is a value between 1 and 6. A picked up die has no face. Model this using the vocabulary* $Face(Die, \{1..6\}, \mathtt{T}), Holds(Die, \mathtt{T}), Throw(Die, \mathtt{T}), Pick(Die, \mathtt{T})$.

Inertial function symbols For an inertial function symbol $F/n + 1 :$, the frame axioms can be greatly simplified.

- introduce $I\_F/n :, C\_F(n + 2)$.
  - We need only one causality predicate, since the value of a function is inertial exactly when no new value is caused.
- The frame rules can be expressed:

$$\left\{ \begin{array}{l} \ldots \\ \forall \bar{x}(F(\bar{x}, 0) = I\_F(\bar{x}) \leftarrow) \\ \forall \bar{x} \forall y \forall t(F(\bar{x}, t + 1) = y \leftarrow C\_F(\bar{x}, y, t)) \\ \forall \bar{x} \forall t(F(\bar{x}, t + 1) = F(\bar{x}, t) \leftarrow \neg \exists y C\_F(\bar{x}, y, t)) \\ \ldots \end{array} \right\}$$

This is a logical representation of an assignment operation.

Warning: when defining a function, declare it always to be a partial function

```
partial F(T1,...,Tn):T
```

Otherwise IDP will assume that it is a total function. This may lead to unexpected inconsistencies because some definitions or axioms are inconsistent with the total function constraint. See, e.g.,

`http://dtai.cs.kuleuven.be/krr/idp-ide/?present=PartialFun`

Symmetrical causation rules In case of contradictory causations, we have seen that CT_P dominates CF_F. I.e., if there is cause for P and a cause for $\neg P$, then P will be caused.

It is possible and sometimes useful to modify the LTC such that in case of contradictory causations, the fluent behaves inertial, that is, it retains its current value. This is achieved by replacing the second rule of the definition of P by the *symmetric causation rule*:

$$\forall t \forall \bar{x}(P(\bar{x}, t+1) \leftarrow CT\_P(\bar{x}, t) \wedge (CF\_P(\bar{x}, t) \Rightarrow P(\bar{x}, t)))$$

Exercise 3.3.8. *Test the summetric causation rule* ***http://dtai.cs.kuleuven.be/krr/idp-ide/?present=Gear-symmetrical***

Event Calculus style of inertial rules There are other ways to express frame axioms. Here is another one:

$$\begin{aligned}\forall \bar{x} \forall t(P(\bar{x}, t) \leftarrow \exists t1 : t1 < t \wedge CT\_P(\bar{x}, t1) \wedge \\ \neg \exists t2(t1 < t2 < t \wedge CF\_P(\bar{x}, t2))\end{aligned}$$

In words, P holds at t it is caused at an earlier time $t1$ and it is not caused false during the interval $]t1, t[$.

This axiom was used in another form of temperal theories called the *event calculus*.

3.3.3 A historical example

The McCarthy and Hayes paper of 1969 led to the development of a new area of logic called *nonmonotonic reasoning*.

By 1980, the logic community had developed several nonmonotonic reasoning formalisms to solve the frame problem. They felt confident.

Then in 1985, two scientists Hanks and McDermot threw a bomb in the community. They showed an extremely simple problem, called the *turkey shooting problem* and demonstrated errors in no less than 3 different main nonmonotonic formalisms that had been specially designed for solving the frame problem. It was a shock.

After this event, a number of people returned to classical logic (including Ray Reiter).

Example: The extended Yale Turkey shooting problem A simple domain exploiting some extra facilities of the LTC:

> *Initially, there is a hunter, a living turkey and an unloaded gun. The hunter can load a gun if it is not loaded. He can always shoot, and if the gun is loaded, the turkey dies. There is an action of waiting which can always be done, and which has no effects. (*) The turkey is dead iff it is not alive. A turkey can start to walk, but only if it is alive, and it can hold still. Any action that kills the turkey stops it from walking.*

The first part of this problem up to (*) corresponds to Hanks and McDermots original Turkey Shooting problem. It came with a scenario as follows: Initially the turkey is alive, the gun unloaded. There are three actions: the hunter loads the gun at time 0, waits at time 1, shoots at time 2. What holds at time 3? A good solution should predict that at 3, the turkey is dead. None of the tested nonmonotonic solutions of the time could produce that result.

A solution in LTC Since there is only one turkey, gun and hunter, we leave these concepts implicit.

There are 4 actions:
- $Load, Shoot$ performed by the hunter
- $Walk, Stop$ performed by the turkey

There are 3 fluents:
- $Alive, Walking, Dead$: fluents of the turkey
- $Loaded$: fluent of the gun

We will define $Dead$ it in terms of $Alive$. The remaining fluents $Alive, Walking, Loaded$ are inertial.

There are two special cases in this example:
- the ramification that getting killed terminates $Walking$;
- $Dead$ is defined in terms of $Alive$.

The LTC theory:

- Initial state:

$$I\_Alive \wedge \neg I\_Loaded$$

- Causal rules:

$$\left\{ \begin{array}{l} \ldots \text{(frame rules for } Loaded, Walking, Alive) \ldots \\ \forall t(CT\_Loaded(t) \leftarrow Load(t)) \\ \forall t(CT\_Walking(t) \leftarrow Walk(t)) \\ \forall t(CF\_Walking(t) \leftarrow Stop(t)) \\ \forall t(CF\_Alive(t) \leftarrow Shoot(t) \wedge Loaded(t)) \\ \forall t(CF\_Walking(t) \leftarrow CF\_Alive(t))) \end{array} \right\}$$

  Notice that we did not express that shooting terminates walking, but the more general rule that action that terminates alive, also terminates walking. When later, the action $TurkeyHeartAttack$ is introduced, its effect on walking is covered by the current ramification rule.

- Preconditions:
$$\forall t(Load(t) \Rightarrow \neg Loaded(t))$$
$$\forall t(Walk(t) \Rightarrow Alive(t))$$
- No-concurrency axiom :...
- Definition of *Dead*
$$\{\ \forall t(Dead(t) \leftarrow \neg Alive(t))\ \}$$

Exercise 3.3.9. *Without no-concurrency axiom, there is a problem if at the same time, the turkey is shot with a loaded gun and starts to walk. Check out what happens and solve this with a relaxed concurrency axiom.*

Exercise 3.3.10. *Adapt the problem such that the hunter can miss the turkey; i.e., make shooting non-deterministic.*

Example 3.3.10. Transmission of signals

> *A sender sends signals to a receiver. Signals may get lost. The receiver receives the signal with certain delay. Signals are received in order of transmission. Each transmitted signal is different.*

There is only one sender and one receiver, so we will not make them explicit.

$$\begin{array}{l}
Type\ Signal \\
Send(Signal, \mathtt{T}) \quad \%\text{action} \\
Receive(Signal, \mathtt{T}) \quad \%\text{action} \\
Lose(Signal, \mathtt{T}) \quad \%\text{action} \\
OnMedium(Signal, \mathtt{T}) \quad \%\text{fluent} \\
\left\{ \begin{array}{l} \ldots \\ \forall s \forall t(CT_OnMedium(s,t) \leftarrow Send(s,t)) \\ \forall s \forall t(CF_OnMedium(s,t) \leftarrow Receive(s,t)) \\ \forall s \forall t(CF_OnMedium(s,t) \leftarrow Lose(s,t)) \end{array} \right\} \\
\%\text{a signal is sent only once} \\
\forall s \forall t \forall t1(Send(s,t) \wedge Send(s,t1) \Rightarrow t = t1) \\
\%\text{Action concurrency:} \\
\forall s \forall t(Receive(s,t) \wedge Lose(s,t) \Rightarrow t = t1) \\
\%\text{Action preconditions:} \\
\forall s \forall t \forall t1(Receive(s,t) \Rightarrow OnMedium(s,t)) \\
\forall s \forall t \forall t1(Lose(s,t) \Rightarrow OnMedium(s,t)) \\
\%\text{Signal ordering is preserved.} \\
\forall s \forall s2 \forall t \forall t2(Send(s,t) \wedge Send(s2,t2) \wedge t2 < t \Rightarrow \\
\qquad \forall t1(Receive(s,t1) \Rightarrow \neg OnMedium(s2,t1))
\end{array}$$

The last formula is not a state or bi-state formula. Strictly spoken, it falls outside the basic LTC. The verification method that we will see in the later section does not work for this specification.

Exercise 3.3.11. *A software package dependency system consists of a dynamic collection of software components each importing a number of required "import" services from other software components and providing a number of "export" services to other software components. E.g., think of a software component as a software library, and of a service as a procedure or function. Libraries import services from and export services to other libraries.*

- *Software components exist in different versions as specified by a version number.*
- *For each software component, the required import and export services are given. Imported and exported services of all versions of the same component are the same and are described by static predicates. The same service may be exported by different components (e.g., two libraries providing two implementations of the same abstract data type).*
- *At each time, each software component should be "installed": each of its import services should be imported from some other component in the system. This induces a dependency relation on the components.*
- *The Component dependency should be acyclic.*
- *There are two actions: to add and to remove a component with some version number.*
- *The system uses a* highest version policy*: each component imports services from the most recent version of a provider component.*
- *When different components offer the same exported service, a component that imports that service has the choice from where it imports it.*

Express this in LTC. [2] *In this problem, there are no inertial predicates.*

3.4 Inference on LTC

For static FO(.) theories we saw that many useful problems can be solved by applying various forms of inference on the domain specification. This is even more so for dynamic FO(.) LTC theories. There is a zoo of problems in various fields of computer science and AI that can be explained in terms of existing and some new forms of inference in LTC. This includes some approaches to verification.

We will give an overview of the many tasks and problems that can be solved on the basis of a LTC.

3.4.1 Applications of Finite Model Generation

Observation: The value of the time type $\mathtt{T}$ can be set to infinite time $\mathbb{N}$ or to finite time $[0, N]$. Most LTC theories have a special property:

- any model $\mathfrak{A}$ with $\mathtt{T}^{\mathfrak{A}} = [0, N]$ can be extended at the tail to a model with infinite time $\mathtt{T}^{\mathfrak{A}} = \mathbb{N}$, and vice versa,
- any model $\mathfrak{A}$ with $\mathtt{T}^{\mathfrak{A}} = \mathbb{N}$ projected on $\mathtt{T}^{\mathfrak{A}} = [0, N]$ yields a model with finite time.

Hence, for such theories, we can study the LTC with infinite time by restricting time to finite intervals.

Exercise 3.4.1. *Check that Peano's theory does not have this property.*

[2] This example is presented in the book of Huth and Ryan.

Applying finite model generation on LTC

- Take a LTC theory, restrict time to some finite interval, add additional desired or undesired propositions, and apply a finite model generator or a finite model expander like IDP.
  - A model is an execution of the system that obeys all our constraints.
  - We get information from a model or from the absence of models.
- A flexible method:
  - Specify the details that you are interested in, and let the solver search for a possible state of affairs. Specify the initial state or vice versa, specify constraints on intermediate or final states, or both. Specify the actions or not. This way, we get insight in many possible behaviours of the system.
  - Add constraints that are the inverse of what the dynamic system should do or of what we expect the system can do. If a model is found, an error has been detected.

To use IDP, the suitable form of inference is model expansion:

- Input: LTC theory T, finite structure $\mathfrak{A}_i$ specifying finite time $[0, N]$, finite domains for all types, and possibly more data.
- Output: a model $\mathfrak{A}$ of T expanding $\mathfrak{A}_i$

To handle the finite domain, T is to be adapted to quantify bi-state formulas φ over the interval $[0, N-1]$, as explained earlier on page 106.

Reducing temporal problems to finite model generation inference AI-problems that can be approached this way:

- *Prediction*: Given an initial state and a sequence of actions, what is the final state
  - Examples: the running example, the Yale Turkey Shooting problem.
- *Postdiction*: Given an observation on some final state, explain in terms of possible initial state and actions.
- Both prediction and postdiction can be non-deterministic due to unknown initial state or nondeterministic actions.
- *AI-planning*: Given initial state and a desired goal, find a plan, a sequence of (potentially concurrent) actions that transforms initial state in a state satisfying the goal proposition.

AI Planning by model expansion The problem with solving a planning problem using finite model expander like IDP is that the input fixes the number of time points. Yet we do not know how many time points will be needed for a planning problem.

Many AI-planning systems do not have this limitation.

For STRIPS it can be shown that in the worst case, an exponential number of time points (hence, actions) is needed, exponential in the number n of fluent atoms of the planning domain.

Proof. sketch of upperbound: With n fluent atoms, we can build 2^n different states. If there is a plan solving a planning goal, there is one without repetition of intermediary states. Hence, if there is a plan there is plan of maximally 2^n long. This is an upperbound. It can be shown to be the worst case lowerbound. ■

Exercise 3.4.2. *Think of a planning problem that takes exponentially many actions; such a planning problem proves the exponential worst case lower bound.*

AI Planning by model expansion A solution for this problem is by iterated calls of model expansion while increasing the time interval in $\mathfrak{A}_i$. In IDP, this is achieved by iterated model expansion calls with an input structure including

$$\texttt{Time} = \{\texttt{0..N}\}$$

for $N = 1, 2, 3, \ldots$ until a plan is found. When a plan is found with this methodology, it will be a shortest plan; if no plan exists, the method does not terminate. This is an effective strategy, certainly to compute the shortest plan. Solvers based of this strategy do well in the AI-planning competition in the category "shortest plan" and have won many of them.

To implement this effectively, it is necessary that systems can incrementally reuse computations for N at $N + 1$. This is a feature that IDP does not (yet) offer.

Example 3.4.1. A classical planning problem: *The towers of Hanoi.*

`http://dtai.cs.kuleuven.be/krr/idp-ide/`

Select "File"
Select "9. Visualisations"
Select "The Towers of Hanoi"

Exercise 3.4.3. *Another classic is the blocks world problem. It is a domain of blocks and a table and one robot that can pick up and put down blocks on the table or on other blocks. To pick up a block, the block must be clear and the robot free. Choose a vocabulary and express in LTC.*

3.4.2 Light weight verification by finite model generation

This section is inspired by the Alloy language and tool:

`http://alloytools.org/`

Alloy is a language and tool for abstract specification and verification of dynamic systems. It was developed at MIT by Daniel Jackson. The language is a syntactic variant of FO. It performs reasoning by bounded model generation. Alloy theories need to specify a maximal size of various types.

The verification problem After building a formal modelling of a domain, we want to analyze it. We expect *emergent* properties of the system: properties that will be satisfied by the described system. Such properties should be entailed by the specification.

A *verification problem* then consists of verifying if such a property is *entailed* by the formal specification.

One frequently useful kind of property that needs to be verified is called an *invariant*. It is a property about a snapshot in time, and the goal is to prove that it is true at each instant of time.

Definition 3.4.1. A state formula $\varphi[t]$ is an *invariant* of a LTC theory T_a if $T_a \models \forall t \varphi[t]$.

Abusing terminology, we sometimes call $\forall t \varphi[t]$ an invariant.

E.g., in the running *Book* example, an expected emergent property is that at time t no book is owned by two persons simultaneously, for each time instant t. That is, we expect that this property is an invariant of the book LTC. Another expected invariant is that a book always has an owner.

However, proving such an emergent property is a non-trivial deductive inference problem. Indeed, recall that T_a is not an FO theory, since it contains the domain closure axiom for time, which is $DCA(\mathbb{N})$ (see Section 2.5.2). This axiom cannot be expressed in FO (this will be shown in Chapter 6). Thus, theorem provers for FO cannot be used for verification from T_a. Worse, it will be shown in Chapter 6 that for any logic in which $DCA(\mathbb{N})$ can be expressed, no complete theorem provers can be built.

For this reason, Alloy focusses on simpler forms of reasoning based on finite model generation. It does not serve to prove emergent properties from the specification. Instead it is used to find errors in specifications.

Guided simulation Beside searching for bugs in a specification, there are many other uses of finite model generation to explore a LTC theory:

- Find an "execution" that violates a desired invariant.
- **Simulate** the dynamic system by generating models.
- Check if a given sequence of actions is possible, and if so, what are the states at different time points?
  Ex. Propose a sequence of request actions in an elevator specification, and search for a finite model. Inspect the intermediate states.
  Ex. **See also : Automatic generation of test-values in C#**

In Alloy, this is called *guided simulation*.

3.4.3 Light weight verification of programs

Light weight verification can be used for analyzing programs. We illustrate the principle.

Example 3.4.2. Translating a program in LTC. The following program GCD computes the greatest common divider of variables n, m. Notice that it is annotated with program points. The purpose will soon become clear.

$P1$ *while* $(n \neq m)\{$
P2 $if\ (n > m)$
then $\{P3\ n = n - m\ \}$
else $\{P4\ m = m - n\ \}$
$\}$
P5

A program is a specification of dynamic processes. As such it can be specified in LTC. Variables are inertial dynamic functions. A function $CurrPP$ specifies the current *program points*. Program instructions modify variables and program points. $CurrPP$ is not really inertial. It changes every time, according to an easy pattern. We define it directly.

$$Type\ PP = \{P1; P2; P3; P4; P5\}$$
$$CurPP(\mathtt{T}) : PP \quad n(\mathtt{T}) : int \quad m(\mathtt{T}) : int$$

$$\left\{\begin{array}{l} n(0) = I\_n \leftarrow \\ \forall t(n(t) = x \leftarrow C\_n(x,t)) \\ \forall t(n(t+1) = n(t) \leftarrow \neg\exists x C\_n(x,t)) \\ m(0) = I\_m \leftarrow \\ \forall t(m(t+1) = x \leftarrow C\_m(x,t)) \\ \forall t(m(t+1) = m(t) \leftarrow \neg\exists x C\_m(x,t)) \\ \\ \forall t(C\_n(n(t) - m(t), t) \leftarrow CurrPP(t, P3)) \\ \forall t(C\_m(m(t) - n(t), t) \leftarrow CurrPP(t, P4)) \end{array}\right\}$$

$$\left\{\begin{array}{l} CurrPP(0, P1) \leftarrow \\ \forall t(CurrPP(t+1, P2) \leftarrow CurrPP(t, P1 \wedge n(t) \neq m(t)) \\ \forall t(CurrPP(t+1, P5) \leftarrow CurrPP(t, P1) \wedge \neg(n(t) \neq m(t)) \\ \forall t(CurrPP(t+1, P3) \leftarrow CurrPP(t, P2) \wedge n(t) > m(t)) \\ \forall t(CurrPP(t+1, P4) \leftarrow CurrPP(t, P2) \wedge \neg(n(t) > m(t)) \\ \forall t(CurrPP(t+1, P1) \leftarrow CurrPP(t, P3) \vee CurrPP(t, P4)) \end{array}\right\}$$

Exercise 3.4.4. *Transform this definition to FO using predicate completion.*

The models of this theory represent the computing processes obtained by running the GCD program. IDP can be applied to compute models on some finite interval. This can be useful for different purposes:

- Compute GCD of given input.
- Compute input from given output.
- Compute input that leads to a specific control flow.
- Search for input such that the outcome is not the GCD of the input.
- Search for input that would lead to a loop.

This is worked out in detail here:

`http://dtai.cs.kuleuven.be/krr/idp-ide/?present=GGD`

(You should understand what happens in this theory.)

Below we sum up potential uses for such theories.

Light weight verification Add an axiom expressing that the program does not satisfy the intended postcondition and apply a finite model generator. If a model is found, the program is certainly not correct. If no model is found, we cannot yet be sure that the program is correct. However, our confidence has grown.

Generation of test values Testing is the most common method for verifying correctness. To find useful and sufficiently exhaustive test-input is important, difficult and time consuming.

Experience shows that many errors are caused by the fact that certain execution flows of the program were not taken into account.
Ex. A flow passing through a missing "else" of an if-test.
Ex. A flow that first passes through the "then" of one if-test, next through the "else" of the next if-test.

Systems are in development that systematically search for test input to test every such different execution flow. After that, the program is ran on these test values to check for correctness.

Several implementations exist, e.g., for Microsofts language C#. Systems for automated test case generation use Constraint Programming or SMT solvers.

Generating a test value for a given chosen control flow is done by generating a constraint on the execution flow.

We illustrate this with the above IDP theory. We formalizing the proposition that the program passes through the *else* instruction during the first iteration, and through the *then* instruction during the second iteration simply as follows:

$$CurPP(2) = P4 \wedge CurPP(5) = P3$$

This is experiment (f) in `http://dtai.cs.kuleuven.be/krr/idp-ide/?present=GGD`

Searching for infinite loops Consider the following proposition:

$$m(0) = m(3) \wedge n(0) = n(3)$$

Notice that 0 is the start and 3 is the time after the first iteration. Suppose that IDP finds a model of the theory and this proposition, for a time interval with at least 4 time points.

What such a model tell us is that there exists an input value for m and n for which the program loops.

A reflection on "declarative programming"

Definition 3.4.2. A program = a description of a class of computer processes, one per input.

Computer languages serve to express programs. Programs are declarative descriptions of computer processes. In a declarative sense, and seen from a slightly abstract level, the above LTC and the program it encodes are *equivalent* in the sense that what they describe is the same: they describe the same processes.

The term "declarative programming" is currently a blurred term. In NL, "declarative" is synonymous to "descriptive". As such *declarative languages* serve to *describe*. But in this respect, all meaningful languages serve to describe, including imperative programming languages which are typically not considered to be declarative. I have no problem to call programming languages "declarative". However, I argue that we should look at what is described and take this seriously.

E.g., a program to compute a function should not be confused with a definition of this function. A logical definition of the GCD function is as follows.

$$\left\{ \begin{array}{c} \forall n \forall m \forall k(GCD(n,m) = k \leftarrow \exists n1(k \times n1 = n) \wedge \exists m1(k \times m1 = m) \wedge \\ \neg \exists v(k < v \wedge \exists n2(v \times n2 = n) \wedge \exists m2(k \times m2 = m)) \end{array} \right\}$$

or recursively:

$$\left\{ \begin{array}{l} \forall n(GCD(n,n) = n \leftarrow n > 0) \\ \forall n \forall m(GCD(n,m) = GCD(n-m,m) \leftarrow n > m > 0) \\ \forall n \forall m(GCD(n,m) = GCD(n,m-n) \leftarrow 0 < n < m) \end{array} \right\}$$

What these definitions describe is the GGD function, the mapping from pairs of natural numbers to numbers. Such a function is an entity of a very different nature than the LTC theory that encodes the GCD program, and hence, than the GCD program.

Exercise 3.4.5. *Insert this definition in the IDP GCD-file and use it to do a light weight verification that the GCD program computes the GCD function.*

3.4.4 Heavy weight verification of invariants

Verification of desired system properties is an important concern of this course. So far, we have only seen light weight methods based on finite model generation which allow us to find bugs in specifications. However, such methods cannot ensure that some property holds for all executions of the modelled system.

Examples of properties to be verified:

- Ex. There is exactly one owner per book:

$$\forall t \forall b \forall p_1 \forall p_2(Owns(p_1,b,t) \wedge Owns(p_2,b,t) \Rightarrow p_1 = p_2).$$

- Ex. When moving, the doors of the train are closed.
- Ex. A traffic light is never green in two directions.
- Ex. It is always possible for the elevator to reach a state with open doors at the ground floor.
- Ex. For each direction, the traffic light eventually becomes green.
- Ex. The system is never in a deadlock.

Looking at these propositions, we see that the first three are propositions that some property about a single state is true at all times. If such a proposition holds, we will call it an invariant of the system. The other propositions are more complex and refer to evolutions of the system. They are more complex to express and more complex to be verified. They will be studied in the next chapter.

Recall that a state formula $\varphi[t]$ is an *invariant* of a LTC theory T_a if $T_a \models \forall t \varphi[t]$.

How to verify that $T_a \models \forall t \varphi[t]$? The first idea that comes to mind is to call a theorem prover to verify $T_a \models \forall t \varphi[t]$. But T_a contains the interpreted type `T` interpreted by $\mathbb{N}$. Thus, theorem provers should be able to reason about the natural numbers. There is no magic in the world; such systems can only reason if they have a *theory* of natural numbers.

Peano's second order theory $\mathtt{Th}(\mathbb{N})$ is such a theory, but it contains a SO induction axiom expressing $DCA(0, S/1)$. However, this is not useful since no good automated SO theorem provers exist.

Instead we could use Peano's FO *arithmetic*, which is an infinite FO theory: instead of the SO induction axiom, it contains the infinitely many instances of the induction schema. There exists contemporary theorem provers supporting inductive proofs using the induction schema. E.g. Coq, Isabelle. Their disadvantage is that none is fully automatic. They all require interaction with the human user to choose the right instances of the induction schema.

There is however a partial solution that can be automated. Assume that for a given LTC theory T_a, we want to prove $\forall t \varphi[t]$, i.e., that the state formula $\varphi[t]$ is an invariant. What instance of the induction schema is needed? An obvious choice would be the instance that uses $\varphi[t]$:

$$\varphi[0] \wedge \forall t(\varphi[t] \Rightarrow \varphi[S(t)]) \Rightarrow \forall t \varphi[t]$$

We denote this instance as $Ind(\varphi[t])$. This sentence is entailed by an LTC theory T_a, since `T` is interpreted by $\mathbb{N}$.

We now develop a methodology for proving invariants $\forall t \varphi[t]$ using $Ind(\varphi[t])$. We will show that this method is not complete (i.e., we cannot prove all invariants with it) but it is useful nevertheless. It has been implemented in many existing systems.

Bistate theories and invariants The method is applicable to LTC theories of the following form:

Definition 3.4.3. We call an LTC theory a *bi-state LTC theory* if it has the following form:

$$T_a = T_{static} \cup T_0 \cup T_s \cup T_t$$

where

- T_{static} is an FO(.) theory of the static symbols,
- T_0 is a set of initial state expressions,
- T_s consists of single state expressions $\forall t \varphi[t]$ or definitions consisting of single state rules,
- T_t consists of bi-state formulas and definitions consisting of bi-state rules.

T_s includes action preconditions and no-concurrency or simultaneity axioms and may include fluent definitions which consist of single state rules. T_t includes the definition of fluents by frame rules and of causality predicates.

Definition 3.4.4. Given any single state or bi-state theory T and numeral n.

The theory denoted $T[n]$ consists of expressions $\varphi[n]$ (rule or FO axiom) for each expression $\forall t\varphi[t]$ in T.

Explained in another way, $T[n]$ is obtained by dropping quantifiers $\forall t$ and substituting n for t in formulas and rules.

The verification method

- Input:
  - a bi-state LTC theory $T_a = T_{static} \cup T_0 \cup T_s \cup T_t$
  - a single state formula $\psi = \varphi[t]$.
- The first step serves to establish the base case of the inductive proof. We construct the following theory:
  - $T_{init} = T_{static} \cup T_0 \cup T_s[0]$

  This theory contains all information of T_a pertaining to the initial state
- We then use a theorem prover to verify:
  - $T_{init} \models \varphi[0]$

  If this call succeeds, T_a entails that φ holds in the initial state. If it fails, report failure.
- The second step serves to prove the inductive step. Construct the following theory:
  - $T_{ind} = T_{static} \cup T_s[0] \cup T_s[1] \cup T_t[0] \cup \{\varphi[0]\}$

  This theory contains all information of T_a that relates two successive states, here represented as 0 and 1.
- We use a theorem prover to verify:
  - $T_{ind} \models \varphi[1]$
- If also this call succeeds, then return that $\varphi[t]$ is an invariant of T_a, otherwise report failure.

The induction axiom is implicitly used in this method, as explained below.

Theorem 3.4.1. *If* $T_{init} \models \varphi[0]$ *and* $T_{ind} \models \varphi[1]$ *then* $T_a \models \forall t\varphi[t]$

Proof. $Ind(\varphi[t])$ is the induction schema instance $\varphi[0] \land \forall t(\varphi[t] \Rightarrow \varphi[S[t]) \Rightarrow \forall t\varphi[t])$.

We have $T_a \models Ind(\varphi[t])$ since `T` is the interpreted type $\mathbb{N}$ for which all induction schema instances hold.

We have $T_a \models \varphi[0]$ since T_{init} is included in T_a.

Since $T_{ind} \models \varphi[1]$, the following holds:

- Then $T_{static} \cup T_s[0] \cup T_s[S(0)] \cup T_t[0] \cup \{\varphi[0]\} \models \varphi[S(0)]$.
- Then $T_{static} \cup T_s[0] \cup T_s[S(0)] \cup T_t[0] \models \varphi[0] \Rightarrow \varphi[S(0)]$.
- Then $T_{static} \cup T_s \cup T_t \models \varphi[0] \Rightarrow \varphi[S(0)]$.
- Then $T_{static} \cup T_s \cup T_t \models \forall t(\varphi[t] \Rightarrow \varphi[S(t)])$. Why? 0 does not occur in $T_{static} \cup T_s \cup T_t$. In classical logic, if $T \models \varphi[c]$ and c is a constant that does not occur in T, then $T \models \forall x\varphi[x]$.
- Then $T_a \models \forall t(\varphi[t] \Rightarrow \varphi[S(t)])$ (T_a includes $T_{static} \cup T_s \cup T_t$)

Now, application of $Ind(\varphi[t])$ yields that $T_a \models \forall t\ \varphi[t]$. ■

An experiment with IDP

`http://dtai.cs.kuleuven.be/krr/idp-ide/?present=BookInvariantsSpass`

On this webpage, the method is manually applied on the running example. It shows the following:

- how to split up `T_Book` in `T_init` and `T_ind`;
- how to apply the theorem prover in IDP to prove the invariant.
  - The theorem prover is SPASS, developed at the max planck institut informatic.
  - SPASS is a FO theoremprover, not FO(.).
  - IDP translates FO(.) to FO, if possible.

The example also illustrates the use of the IDP procedure

isinvariant(LTC-theory,invariant)

that implements the heavy weight invariant verification method.

Sometimes it is useful to prove invariants in specific domain. For this IDP provides the method

isinvariant(theory T,invariants I,structure $\mathfrak{A}$*)*

$\mathfrak{A}$ is a structure specifying a finite domain for static types. IDP will perform the invariant verification method, except that it will limit the search to structures that expand $\mathfrak{A}$.

Such an example is found at: `http://dtai.cs.kuleuven.be/krr/idp-ide/?present=BookInvariant`

Limitations of the method The method has several weaknesses explained below.

The method is not complete Two of the cases where it may fail to prove the invariance of an invariant:

- If the invariants are not sufficiently strong: see the light switch example.
- If there are other integer-valued types. E.g., for proving correctness of a program with integer-valued variables (such as the GCD program). In that case, other instances of the induction schema are needed to prove properties of these integer valued types and relations.

The invariant should be strong enough The above methodology may not work if the invariant is not strong enough. It may be that a set of invariants $\forall t \varphi[t]$ are mutually dependent on each other. It might be impossible to prove the invariance of each in isolation of the others.

We illustrate this in a light switching example.
`http://dtai.cs.kuleuven.be/krr/idp-ide/?present=Light`

The specification:

$$\left\{\begin{array}{l}\ldots \text{frame rules}\\ \forall t(CT\_On(t) \leftarrow \neg On(t))\\ \forall t(CF\_On(t) \leftarrow On(t))\\ \forall t(CT\_Off(t) \leftarrow \neg Off(t))\\ \forall t(CF\_Off(t) \leftarrow Off(t))\end{array}\right\}$$
$$On(0) \wedge \neg Off(0)$$

Two invariants of this theory are the following:

$$\forall t\ On(t) \vee Off(t)$$
$$\forall t\ \neg On(t) \vee \neg Off(t)$$

Remarkably, the method for proving invariance fails to prove invariance of either of these formulas. Suppose we want to prove invariance of $\forall t\ On(t) \vee Off(t)$:

- Step 1 succeeds: $On(0) \vee Off(0)$ is entailed by $On(0) \wedge \neg Off(0)$.
- Step 2 fails. Indeed, one initial state that satisfies this invariant is where On and Off hold at 0. It then follows from T_{ind} that in the successive state 1, it holds that On and Off are both false, hence the invariant is broken.

Actually, it is not difficult to see that $On(1) \vee Off(1)$ follows from $\neg On(0) \vee \neg Off(0)$ and vice versa, that $\neg On(1) \vee \neg Off(1)$ follows from $On(0) \vee Off(0)$. Thereofore , it is a piece of cake to prove that the conjunction $\forall t(On(t) \vee Off(t)) \wedge (\neg On(t) \vee \neg Off(t))$ is an invariant of this LTC. So, while the method is strong enough to prove the conjunction of the two invariants, it is not strong enough to prove the two invariants separately.

Invariants that cannot be expressed in FO To see that FO does not always suffice to express all invariants of a dynamic system, consider a dynamic system in which at each time point, an object moves around in some graph $G/2$; $Pos(v, t)$ means that at time t the object is at vertex v. Its initial position is the vertex A ($Pos(0, A)$). At each time point, the object may stay where it is or move to an adjacent vertex. What is the invariant of this system? It is that

at each time t, the object is at a vertex that is *reachable* from A in graph G. Stated differently, at time, there is a position from A to its current position v through G.

As we shall see in Chapter 7, this invariant cannot be expressed in FO.

Exercise 3.4.6. *Express this dynamic system in LTC. Express the invariant using FO(ID), using an inductive definition and prove the invariant using IDP.*

Example 3.4.3. A similar problem occurs in the gearwheel example. There, an invariant is that all pairs of gearwheels that are connected directly or indirectly, are in the same state (either turning or not).

`http://dtai.cs.kuleuven.be/krr/idp-ide/?present=GearInvariant`

This example illustrates how to prove invariants involving inductively defined predicates.

Some verifiable properties are not invariants. Consider the following propositions:
Ex. The elevator can always reach the state that it is on the ground floor with open doors (but does not need to).
Ex. At all times, if the system is $Busy$, then, after finite time, it will return to the state $Wait$.

These propositions are about *evolution* of the system, not about a *state*. They might be expressible in FO, but we have no *method* to prove them.

To express properties of evolution of the system , we introduce temporal logic in next chapter.

3.4.5 Combining light and heavy weight verification

`http://dtai.cs.kuleuven.be/krr/idp-ide/?present=BookConcurrency`

In this example, a strategy is sketched to combine the different verification methods, to refine and correct the no-concurrency axiom of a simple example.

An illustration: BibSys

- A simple information system for the management of a library.
- BibSys consists of:
  - A database with following relations:
    - $InHoldings(x)$: book x is owned by the bib.
    - $OnLoan(b, p)$: book b is lent out to person p.
    - $Available(b)$: book x is available in the bib.
  - Transactions in BibSys.
    - loan of a book
    - return of a book
    - purchase of a book
    - deletion of a book.

Not every database instance reflects a correct state of affairs of the BibSys system.

- E.g., a book on loan must be in the holdings of the bib.
- E.g., a book cannot be on loan and available.

BibSys - integrity constraints

- $E1$: A book is owned by the bib if and only if it is on loan to some person or it is available: Formally:
$$\forall x(InHoldings(x) \Leftrightarrow Available(x) \vee \exists u OnLoan(x,u))]$$
- $E2$: No book is simultaneously available and on loan. Formally:
$$\forall x(InHoldings(x) \Rightarrow \neg\exists u OnLoan(x,u) \vee \neg Available(x)]$$
- $E3$: No book is lent out to more than one person. Formally:
$$\begin{array}{l}\forall x(\ InHoldings(x) \Rightarrow \\ \quad \neg\exists u \exists w(u \neq w \wedge OnLoan(x,u) \wedge OnLoan(x,w))\end{array}$$

Transactions

A transaction is a program that executes a number of different logically coherent operations on the database.

- E.g., moving a book from *Available* to *OnLoan*. .

Correct transactions preserve integrity constraints.

BibSys as dynamic system BibSys, like other database applications, is a dynamic system.

- Database relations as fluents
- Transactions as actions
- Integrity constraints as invariants

We model BibSys in LTC, using action predicates:

- $Borrows(p,b,t)$: person p borrows book b at time t.
- $Returns(p,b,t)$: person p returns book b at time t.
- $Remove(b,t)$: book b is removed from library at t.
- $Add(b,t)$: book b is added to library at t.

The theory, invariants are expressed in the following IDP-file: `http://dtai.cs.kuleuven.be/krr/idp-ide/?present=BibSys`

Use IDP to:

- Solve a planning problem
- Use light weight verification to refine the no-concurrency axiom
- Simulate BibSys
- Verify the invariants with the context-based invariance proof method.

Exercise 3.4.7. *Notice that the simulation of BibSys in the IDP system comes very close to an actual execution of this system. Think of what we are missing in IDP to use the BibSys theory to actually run the BibSys system. That is, what should be added/modified so that we can actually have a working BibSys software system to manage the library, on the basis of the logic theory .*

3.4.6 Progression inference and (Interactive) simulation inference

Definition 3.4.5. Given an LTC-vocabulary Σ_a, we define its *state vocabulary* Σ_a^s as the set of symbols obtained from Σ_a by dropping all time arguments.

E.g., $Owns(Person, Book, \mathtt{T})$ becomes $Owns(Person, Book)$.

Structures of Σ_a^s represent snapshots of the LTC world.

A finite sequence $\langle \mathfrak{A}_0^s, \ldots, \mathfrak{A}_n^s \rangle$ of Σ_a^s-structures with the same domain corresponds one to one to a Σ-structure $\mathfrak{A}$ where $\mathtt{T}^{\mathfrak{A}}$ is $[0, n]$ such that for each $t \in \mathtt{T}^{\mathfrak{A}}$: the state at time t in $\mathfrak{A}$ is as given by $\mathfrak{A}_t^s$.

A similar correspondence holds for infinite sequences $\langle \mathfrak{A}_0^s, \ldots \rangle$ and Σ-structure $\mathfrak{A}$ where $\mathtt{T}^{\mathfrak{A}} = \mathbb{N}$.

Definition 3.4.6. *Progression inference* is the problem:

- input: a bi-state LTC theory T_a over LTC vocabulary Σ_a and a Σ_a^s-structure $\mathfrak{A}_i^s$
- output: a Σ_a^s structure $\mathfrak{A}_o^s$ such that $\langle \mathfrak{A}_i^s, \mathfrak{A}_o^s \rangle$ corresponds to a model $\mathfrak{A}$ of T_a with $Time^{\mathfrak{A}} = \{0, 1\}$

Progression inference has been implemented in IDP but only works for bi-state LTC theories T_a that consist of:

- Time: `Start:Time` and `Next(Time):Time`
- initial state axioms in `Start`
- state and bi-state LTC rules and axioms: universally quantified state and bi-state formulas and rules using the successor function `Next(Time):Time`
  - $\forall t \varphi[t]$,
  - or definitions with rules of the form
    - $\forall t \ldots (P(\ldots, t) \leftarrow \varphi[t])$ or
    - $\forall t \ldots (P(\ldots, Next(t)) \leftarrow \varphi[t])$.

    where $\varphi[t]$ is a state or bi-state formula.

An application of progression: interactive simulation By iteratively applying progression inference, systems can be simulated with or without interaction.

Algorithm(T_a,$\mathfrak{A}$)
  T_a: LTC theory
  $\mathfrak{A}$: a Σ_a^s-structure specifying initial state
1) Generate a set of initial states allowed by the theory state axioms and the initial state axioms.
2) Loop:
    Let the user select a state from the given set of states.
    Set the selected state as "current state".
    Apply progression inference on "current state".
    Present possible successor states to the user.
    Go to 2.

Interactive simulation: a (toy) implementation in IDP For an interactive simulation of the pacman problem in IDP, see:

`https://dtai.cs.kuleuven.be/krr/idp-ide/?chapter=intro/9-IDPD3`

In the webinterface only a few steps can be executed due to a resource consumption limitation on the server. To execute well, use a local installation of the IDP web-IDE on your computer.

The same LTC specification was used for:
- interactive simulation inference: interactively run a packman scenario;
- optimization inference: search for shortest path taking all gold.

This is an illustration of the KB-paradigm in the context of dynamic worlds.

3.5 LogicBlox: software by "running" specifications

Can you imagine that one day large transaction software systems are built as LTC-like logical specifications that are used by performing "execution inference"? I am aware of at least one quite successful American company that is actually doing this: *LogicBlox*.

`http://www.logicblox.com/`

The company developed a logic based system to designed to implement large scale standard transaction-based software applications.

- book keeping, stock management, financial management, ...

A short summary of the system follows explaining the system in terms of concepts that we have seen earlier in this course in the context of a sales management application.

Figure 3.1 displays a form allowing a user to place an order, and the corresponding rule that registers the order. The context is that an earlier action was performed that created this form. To place an order, the user specifies:
- a product,

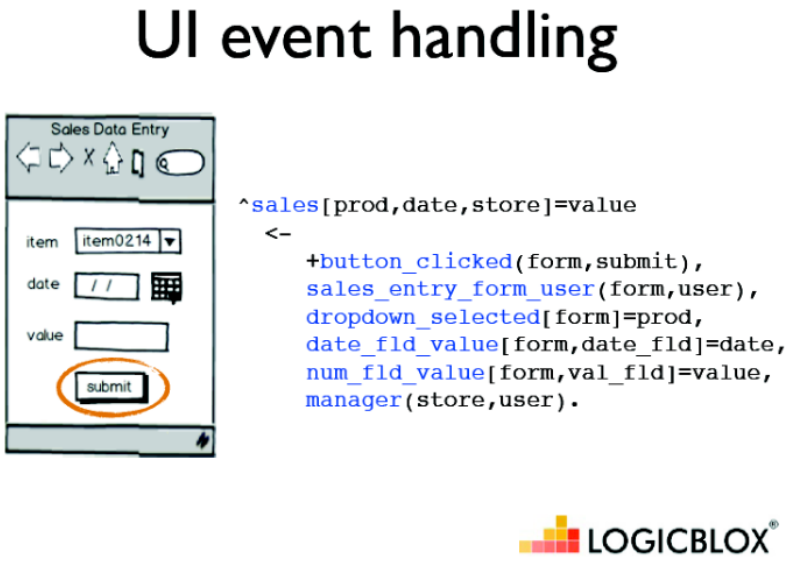


Figure 3.1: A rule of LogicBlox theory for sales management

– a date (of delivery), and
– the required value of that product.

The displayed rule is then triggered when the user clicks the submit button on the form.

To give an idea of what the rule means, we translate it in LTC. Essentially, this is done by turning this in a bistate rule:

$$\left\{\begin{array}{l} \ldots \\ \forall prod\ \forall date\ \forall store\ \forall value\ \forall form\ \forall t \\ C\_sales(prod, date, store, value, t+1) \leftarrow \\ \quad \textbf{button\_clicked}(\textbf{form}, \textbf{submit}, \textbf{t}) \wedge \\ \quad sales\_entry\_form\_user(form, user, t) \wedge \\ \quad dropdown\_selected(form, t) = prod \wedge \\ \quad date\_fld\_value(form, date\_fld, t) = date \wedge \\ \quad num\_fld\_value(form, val\_fld, t) = value \wedge \\ \quad \exists store : manager(store, user, t). \\ \ldots \end{array}\right\}$$

This is an effect rule (hence C_sales), specifying an assignment of $value$ to the fluent function term $sales(prod, date, store)$

- $submit, date\_fld, val\_fld$ are constants identifying fields or buttons in the window;
- $form, user, prod, date, value, store, (t)$ are variables representing the GUI window identifier, the user that is logged in, the product, date, and value filled in by the user on the form.
- **button_clicked**(**form**, **submit**, **t**) : a so-called **pulse atom**; an non-inertial action predicate true when the user clicks the *submit* button.

LogicBlox theories describe dynamic worlds. They are similar to Linear Time Calculus, except that:

- Time is implicit, inertia is built-in (as in Event-B, the system seen in the next Chapters).
- Rules express causal rules or definitional rules.
- Distinction is made between *internal* and *external* fluents. *Internal* fluents describe internal state; e.g., $sales, manager$. *External* fluents are linked with the environment; e.g., $button\_clicked, sales\_entry\_form\_user, dropdown\_selected, date\_fld\_value, num\_fld\_value$.

Running this LogicBlox program is done by *simulation/execution inference*: iterated progression inference.

A progression step is triggered when the *pulse* atom $button\_clicked(form, submit)$ becomes true. The system then computes a *transaction* the results in an update of the state of the system. Once the update is computed, the updated state is stored persistently till the next transaction. The system provides standard transaction management on the cloud. The system scales and is used in large retail chains.

Example 3.5.1. An example in LTC is given to explain the principles underlying LogicBlox. The system is a GUI with the following fields:

- Two input drop down fields called $fld1, fld2$ in which the user can select a character from A to Z.
- Two buttons: "add" to add tuples to a graph, "quit" to quit.
- An output field $Output/1$ for text.

Internal fluents $G/3, Reaches/3$ represent a graph and its transitive closure.

External fluents are:

- $Start$ is a pulse predicate made true when the user calls the LogicBlox command "Start" in the LogicBlox command line. This triggers the form.
- $form\_user, fld\_value, Output$ are external predicates to represent respectively whether the form is active, the current values of the fields of the form, and the current value of an output field on the form.
- $button\_clicked$ is an external pulse predicate that initiates a progression inference when "add" or "quit" is clicked.

The LTC that expresses this dynamic system:

$$T = \left\{ \begin{array}{l} \ldots \\ \forall t(C\_form\_user(t+1) \leftarrow Start(t)). \\ \forall xyt(C\_G(x,y,t+1) \leftarrow button\_clicked(add,t) \wedge \\ \qquad\qquad fld\_value(fld1,x,t) \wedge \\ \qquad\qquad fld\_value(fld2,y,t)). \\ \forall t(Output(\text{"}Error!\text{"},t) \leftarrow Reaches(A,Z,t)). \\ \forall t(CN\_form\_user(t+1) \leftarrow button\_clicked(quit,t)). \\ \forall xyt(Reaches(x,y,t) \leftarrow G(x,y,t)). \\ \forall xyt(Reaches(x,y,t) \leftarrow G(x,z,t) \wedge Reaches(z,y,t)). \end{array} \right\}$$

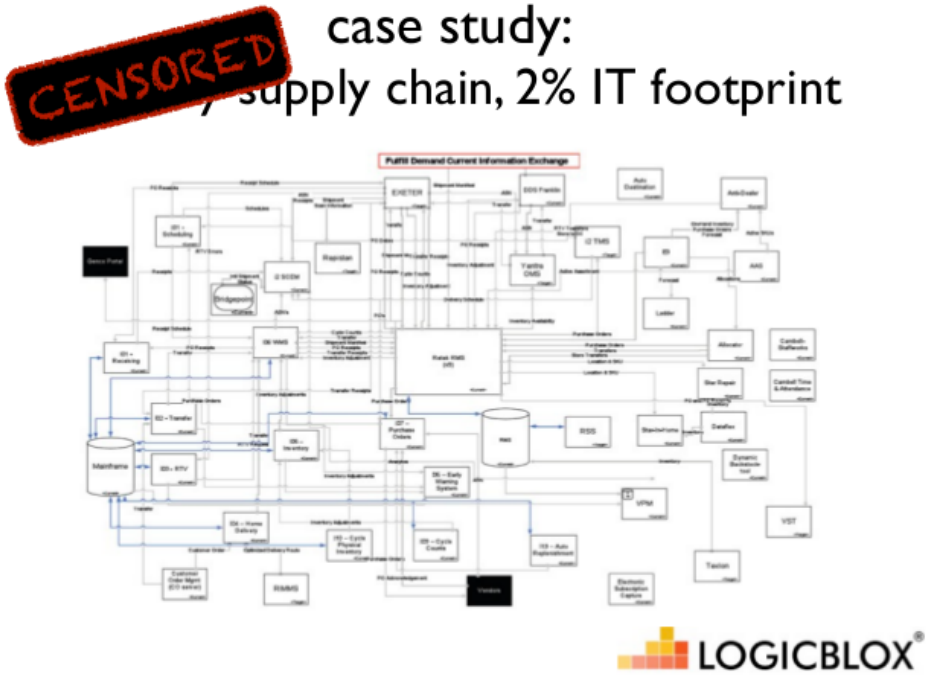


Figure 3.2: The company hairball

Iterated progression runs the application:

- It starts from the persistent current state $\mathfrak{A}$ of the internal predicates $G, Reaches$ and the current external state (whether the form is alive, and if it is alive, what are the selected values of the fields).
- When a pulse predicate becomes true by action of the user, a progression inference step is executed on $T, \mathfrak{A}$. In case of a *Start* event, $form\_user$ is initiated; "quit" terminates the same predicate, and "add" causes G to be extended with the current content x, y of the fields $fld1, fld2$; *Reach* and potentially $Output(\text{“}Error!\text{”})$ are updated accordingly.
- After the new state is computed, the persistent internal state and the external state is updated. E.g., if $Output(\text{“}Error!\text{”})$ is true in the new state, "Error" is printed.

LogicBlox is an *executable, state-based workflow language*:
– a theory specifies a workflow;
– the workflow is executed through iterated progression inference.

One of the ambitions of LogicBlox is to tackle the "company hairball" of the kind shown in Figure 3.2. With the hairball, they refer to the often extremely complex bunch of interacting and interrelated software packages and programs that run company applications. The hairball is often leading to terrible maintenance problems. LogicBlox aims to resolve this problem by using its single logical language for almost every task.

LogicBlox claims that they achieve major benefits on the level of development and maintenance: a factor 50× compared to standard Java development for large projects. E.g., LogicBlox com-

Figure 3.3: Users of LogicBlox

pany implemented the stock management of Walgreens with 2 people. This is a big achievement since Walgreens is a large company. It is the largest drug retailing chain in the United States.

Some clients are represented in Figure 3.3.

Final remarks: Using formal specifications for verification Two short recent articles in Communications of the ACM about the usefulness of formal specifications of dynamic systems:

- `http://dl.acm.org/citation.cfm?id=2736348&CFID=722998236&CFTOKEN=28079324`

  Leslie Lamport author of the first works for Microsoft Research, is developer of the TLA+ system, was original developer of LaTeX.

- `http://dl.acm.org/citation.cfm?id=2699417&CFID=722998236&CFTOKEN=28079324`

These articles argue for the usefulness of formal methods for verification and designing systems. An aspect not covered in them is the use of formal specifications to solve problems, perform tasks, or run systems.

3.6 Important for the exam

```
Big questions:
- demonstrate and discuss LTC in a basic example scenario, e.g.,
the Turkey shooting problem.
- discuss different forms of inference in the context of dynamic systems
- heavy weight verification and proof of correctness

Understanding of the concepts and methodology of LTC:
- what the naive representation of Section 1 misses.
```

- the frame problem
- inertia, causation, actions,
- extensions : definitions, non-determinism,
- link with Strips language

Inference:
- different basic forms of inference of use in various applications.
- light weight verification
- light weight verification of programs.
- heavy weight verification of invariants for bistate LTC.
- progression inference and basic understanding how it is used in LogicBlox

There will be no detailed questions about LogicBlox.

Chapter 4

Verification by model checking

4.1 Definition of model checking

Formal verification tools for dynamic systems comprise three parts:

- a language (or system) for dynamic system specification;
- a language for describing propositions to be verified;
- one or more inference methods to establish whether the described dynamic system satisfies the propositions to be verified.

Ex. In the previous chapter, we used bistate LTC in FO(.) for dynamic system specification, single state formulas in FO(.) for goals and invariants, and we use IDP's inference tools for reasoning and problem solving.

In practice, there is often a big difference between dynamic system specification and the propositions one wants to verify. System specification is about transitions, causality, inertia, preconditions, concurrency. System properties to be verified focus on single state properties (invariants) or properties of possible evolutions of the system, such as fairness and deadlock. As such, quite a few systems use different languages for the task of system specification and property specification.

Languages used for *dynamic system specification* are:

- Classical logic or extensions of it as in Linear Time Calculus, Situation Calculus, Alloy (variant of predicate logic)
- Z, specification language using set-theoretical notations
- Special purpose languages such as Programming languages, Petri-nets, abstract state machines, B, Event-B, CSP (Communicuating Sequential Processes), TLA+, etc.

Languages for describing *system properties*:

- Classical logic

- Temporal logic : **Linear Time Logic (LTL), Computation Tree Logic (CTL)**

The focus of this chapter is on the temporal logics for system property specification : LTL and CTL. To experiment with these languages, we will use the ProB tool. This tool supports the Event-B language for dynamic system specification and LTL and CTL for system property specification. It supports various forms of inference for these. Therefore, this chapter contains also a brief introduction to Event-B.

Model Checking In the field of (temporal) model checking, the following approach is taken.

- The dynamic system is specified as a *transition structure* $\mathcal{M}$. This is a transition graph between labeled states. Its definition follows. (see earlier on page 81). Various languages have been designed to describe transition structures. Here, the transition model will be specified in the Event-B language of the ProB system.
- System properties represented by temporal logic formulas. We see three logics: Linear Time Logic (LTL) and Computation Tree Logic (CTL), and their generalization CTL*.
- The verification inference method is model checking. The inference problem that it solves is defined as follows:

  Model checking inference problem:
  input: $\mathcal{M}$, state s in $\mathcal{M}$, temporal logic formula φ
  output: $(\mathcal{M}, s \models \varphi)$.

  In words, the inference returns **t** if $\mathcal{M}, s \models \varphi$ and **f** otherwise.

Transition structures

Definition 4.1.1. A *transition structure* $\mathcal{M} = \langle S, \rightarrow, L \rangle$ of a propositional vocabulary σ of propositional symbols consists of:

- S: the set of states;
- $\rightarrow$: a binary *transition* relation on S;
- $L : S \rightarrow 2^{\sigma}$: a mapping from states to sets of propositions, representing those true in the state.

We write $s_1 \rightarrow s_2$ to denote thatt there is a transition from state s_1 to state s_2.

For each state s, $L(s)$ is a structure of σ representing the snapshot of the world at state s. E.g, for $\sigma = \{p, q, r\}$, the value $L(s) = \{p, q\}$ represents that p, q are true and r is false in state s.

A running example is presented in Figure 4.1.

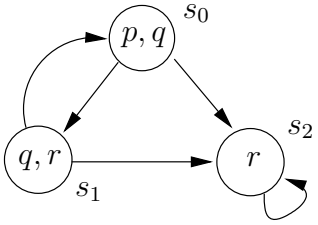


Figure 4.1: A transition structure

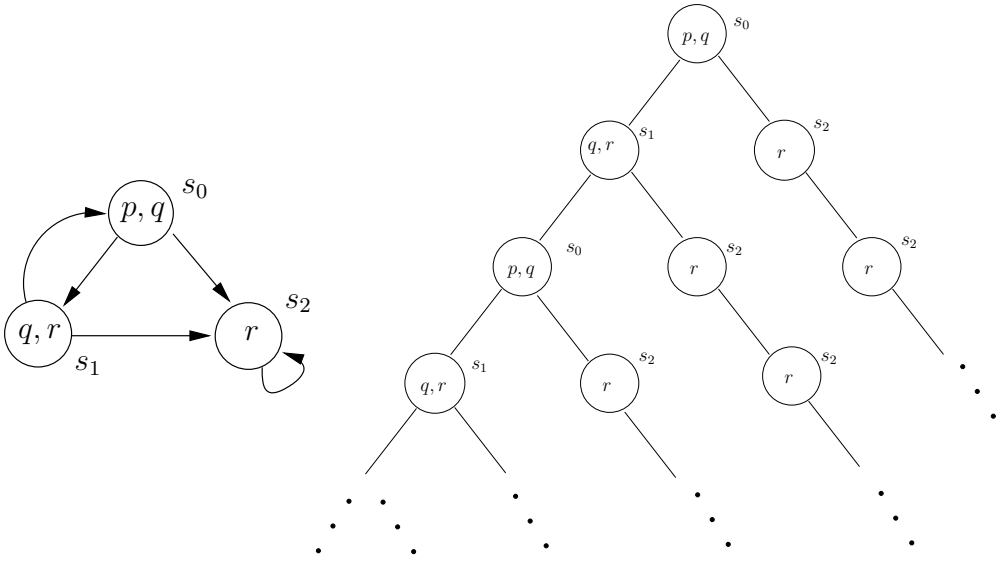


Figure 4.2: A transition structure and its unwinded state tree in s_0.

A path in a transition structure is a representation of a (linear) evolution of the system. In this respect it corresponds to a linear time structure of the LTC. Hence, a transition structure can be seen as a compact representation of a set of linear time structures, namely the set of its paths.

Given an initial state s, a transition structure can be "unwinded" from s as a tree of states. Paths from the root s in this tree represent the possible evolutions from s of the dynamic system expressed by $\mathcal{M}$. Such paths fullfill the same role as structures of an LTC theory. Figure 4.2 shows the unwinding of the running example from state s_0.

Several sorts of extensions of transition systems are in use. Some extensions include one or a set of initial states. In some, edges of the transition relation are labeled by actions or events. Event-B describes transition structures with multiple initial states and transitions labeled with *events*.

The concept of transition structure is strongly related to well-known concepts from theoretical computer science such as finite state machines (FSM) and Automata. E.g, an automaton is a transition system extended with an initial state s_0, a set $\mathcal{A}$ of accept states and labeled edges. It *accepts* a finite string $a_1 \dots a_n$ if this is the sequence of labels on edges in a path $s_0 \xrightarrow{a_1} \dots \xrightarrow{a_n} s_n$ where $s_n \in \mathcal{A}$.

Remark 4.1.1. In the text that follows, the theory of transition structures and temporal logics LTL and CTL is developed for propositional vocabularies only. This is for simplicity reasons.

```
MACHINE Book
Sets
   Person={Bob, John, Mary};
   Book={B1,B2,B3}
VARIABLES Owns
INVARIANT
   Owns:  Person Book &
   !b, p1.((p1,b):owns => !  p2.  ((p2,b):owns => p1=p2))
INITIALISATION Owns:= { (Bob,B1), (John,B2), (Mary,B3)}
OPERATIONS
   Give(g,b,r) = PRE (g,b):Owns & r:Person THEN
      Owns = (Owns - {(g,b)}) \/ {(r,b)}
   END;
END
```

Figure 4.3: The running example in ProB

However, the ProB system supports predicate versions of the Event-B language and of LTL and CTL. In examples and exercises, we will sometimes use these predicate vocabularies both for system description and temporal logic formulas.

4.1.1 ProB

ProB is an inference system developed at U. Dusseldorf by Prof. Michael Leuschel (a former PhD of DTAI). The system supports many languages. For dynamic system specification, it supports Event-B but also other languages, e.g., TLA+, Alloy,.... It supports LTL and CTL for specificatin of process properties. ProB als supports fragments of second order and higher order logic.

The inference methods of ProB are for LTL and CTL model checking and further more, several of those provided by IDP for LTC. The core of ProB is a constraint solver for a very extended language including higher order features. ProB does not support inductive definitions.

A specification of the running example in ProB is given in Figure 4.3.

Some explanation:

- Machines are theories.
- The Event-B language distinguishes between *variables* (=inertial fluents) and *events* (=actions).
- Its syntax is set-based: `:` denotes the element relation $\in$. E.g., `(g,b):Owns` expresses the same as `Owns(g,b)` in FO. The symbol \/ in the assigment to `Owns` represents set union, and the symbol $-$ represent set difference.
- The expression `Person Book` denotes the set of relations between `Person` and `Book`.
- The syntax is action oriented. An action, called an event or an operation, is expressed in one block of the form 'event(parameters) = '`PRE` < precondition > `THEN` < effects >". The figure illustrates this for the event `Give(g,b,r)`.

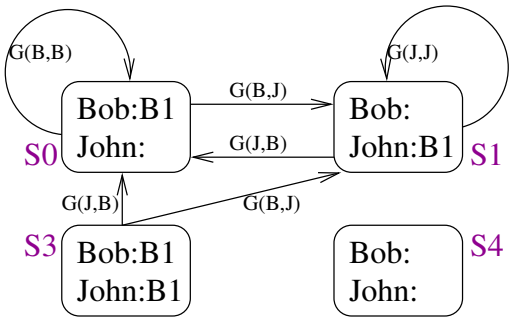


Figure 4.4: The labeled transition graph of `Machine Book`

- Effects are expressed as assignments. They correspond to causations in LTC.
- Inertia is implicit. It is an implicature of the logic. Variables do not change value over time unless an event takes place that assigns a value to them.

The transition structure characterised by an Event-B machine is *action labeled*: transition edges are labeled by an action. Such structures assume that at each transition a unique action/event occurs. They exclude concurrency of actions. As such, non-concurrency of events is built-in in the language. This is an implicit assumption of the Event-B language.

An advantage of the action labeled transition structures is that they are more compact than the unlabeled transition structure. This is because different states which differ only by the action that occurs in it are compressed in one state, and the actions are moved to labels of outgoing edges.

The transition structure for a simplified version of the Event-B Machine Book of Figure 4.3 that assumes `Person={Bob, John}; Book={B1}` is displayed in Figure 4.4.

ProB is introduced in the exercise sessions of this course.

Transition structures modelled by LTC Also LTC theories can be used to characterise transition structures. Specifically, for each combination of a bi-state LTC theory T_a containting the no-concurrency axiom and an input structure $\mathfrak{A}_i$ interpreting all non-time types of T_a, there is unique action-labeled transition structure. This transition structure is equivalent with the combination of T_a and $\mathfrak{A}_i$ in the sense that a one to one correspondence exist between expansions of $\mathfrak{A}_i$ satisfying T_a, and paths of the transition structure starting at initial states.

A pair of a bistate LTC theory and such an input $\mathfrak{A}_i$ induces an equivalent transition structure. Let us call this the *state transition graph of T_a in $\mathfrak{A}_i$*.

Example 4.1.1. The transition structure of Figure 4.4 is the state transition graph of the LTC theory T_{Book} of Chapter 3 in the context of an input structure $\mathfrak{A}_i$ where $Person^{\mathfrak{A}_i} = \{Bob, John\}$ and $Book^{\mathfrak{A}_i} = \{B1\}$.

Recall that a type structure $\mathfrak{T}$ for some set of base type symbols, is an assignment of domains $t^{\mathfrak{T}}$ to every type t in that set.

Exercise 4.1.1. *Think of a way to compute the action labeled state transition graph for a bistate LTC theory in an input structure $\mathfrak{A}_i$ using the forms of inference for LTC seen in the previous Chapter.*

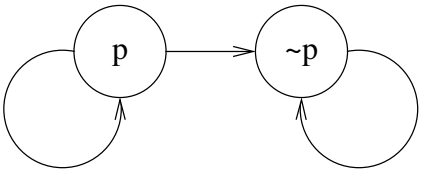


Figure 4.5: A two state transition structure

4.2 Linear-Time Logic LTL

4.2.1 Syntax and semantics

In temporal logic, time is implicit (unlike Linear Time Calculus). Temporal logic propositions do not contain explicit time terms and are evaluated relative to a state; they may be true in one state and false in another. Temporal logic propositions may talk about evolution in time and about the future or past using temporal connectives such as *always in the future, sometimes in the future, until, ...*

We will see two logics implementing different sorts of time topologies.

- Linear Time Logic (LTL) takes a linear topology for time. It serves to specify properties on a single path of the transition structure.

- Computation Tree Logic (CTL) takes a branching time topology. It serves to specify properties about multiple paths, i.e., different evolutions of the system. E.g., in CTL, we can express that some state has a path in which p is always true and also a path in which $\neg p$ is true in the future. This is not a contradiction. E.g., we can express that there is a path in which p is always true although at each time point, it is possible that p becomes false in the future. This means that although p is true at each state of the path, the state at each time point on the path has an alternative future in which p eventually becomes false.

Example 4.2.1. For an example of a structure where the proposition in the previous sentence is true, consider the transition structure in Figure 4.5. The loop at the left through state p has the desired property: p is true at each time point; however, at each time point there is an alternative future off the path in which p becomes false at the next state. This proposition cannot be expressed in LTC nor in LTL because it is not possible there to refer to alternative futures of the same state. It can be expressed in CTL thanks to its branching time topology.

Remark 4.2.1. Do not confuse LTC and LTL.

- LTL : Linear Time **Logic**: a *propositional* logic with temporal modal operators.

- LTC : Linear Time **Calculus**: an LTC theory is a theory in an extension of FO expressing a dynamic domain following the *methodology* explained in Chapter 3.

We now proceed to formally define the logic LTL.

Syntax of LTL

Definition 4.2.1. The syntax of LTL-sentences (over Σ) is defined by the following BNF (Backus Naur form):

$$\varphi ::= \begin{cases} \bot \mid \top \mid p \ \ (\text{where } p \in \Sigma) \mid \\ (\neg\varphi) \mid (\varphi \wedge \varphi) \mid (\varphi \vee \varphi) \mid (\varphi \Rightarrow \varphi) \mid \\ (\mathrm{X}\,\varphi) \mid (\mathrm{F}\,\varphi) \mid (\mathrm{G}\,\varphi) \mid (\varphi\,\mathrm{U}\,\varphi) \mid (\varphi\,\mathrm{W}\,\varphi) \mid (\varphi\,\mathrm{R}\,\varphi) \end{cases}$$

The temporal connectives used are: $\mathrm{X}, \mathrm{F}, \mathrm{G}, \mathrm{U}, \mathrm{W}, \mathrm{R}$. The Temporal connectives of LTL can be nested. E.g., $\mathrm{G}\,\mathrm{F}\,p$ means "p is true infinitely often in the future".

Informal semantics of LTL connectives An LTL formula is evaluated in a path of $\mathcal{M}$.

- $\bot$: false, $\top$: true
- $(\mathrm{X}\,\phi)$: ϕ is true in the *neXt* state;
- $(\mathrm{F}\,\phi)$: ϕ is true in some *Future* state, starting from now.;
- $(\mathrm{G}\,\phi)$: ϕ is *Globally* true. i.e., in all future states of the path, starting from now;
- $(\psi\,\mathrm{U}\,\phi)$: ψ is true *Until* ϕ becomes true. More precisely, in some future state starting from now, ϕ is true and in all preceding states starting from now, ψ is true in the current state. This is expression is satisfied in a path in which ϕ is true in the first state.
- $(\psi\,\mathrm{W}\,\phi)$: *Weak-Until*: the same as U, except that it is satisfied as well if ψ remains true for ever.
- $(\psi\,\mathrm{R}\,\phi)$: ψ *Releases* ϕ: ϕ is true at least until in a current or future state in which ψ is true.

R is similar to W except that the roles of arguments is exchanged and for $\psi\,\mathrm{W}\,\varphi$, ψ needs to remain true until the state preceding the state satisfying φ, while for $\varphi\,\mathrm{R}\,\psi$, ψ needs to remain true until and including the state satisfying φ. As such it holds that $\psi\,\mathrm{R}\,\varphi$ and $\varphi\,\mathrm{W}(\varphi \wedge \psi)$ are equivalent.

LTL connectives do not mean the same as in natural language as the following example shows.

$$\mathit{smoke}\,\mathrm{U}\,\mathit{sick} = \text{I will smoke until I get sick?}$$

Not really. It means that there will come a time that I am sick and before that time, I will smoke. I might go on smoking afterwards.

A more correct translation of the natural language statement is $(\mathit{smoke} \wedge \neg\mathit{sick})\,\mathrm{W}(\mathit{sick} \wedge \mathrm{G}\,\neg\mathit{smoke})$.

In LTL the future includes the present time. As a consequence, valid statements, true in each path are:

- $\mathrm{G}\, p \Rightarrow p$
- $p \Rightarrow q \,\mathrm{U}\, p$
- $p \Rightarrow \mathrm{F}\, p$

Binding conventions of LTL are:

- Unary connectives bind most tightly.
- Then in order $\mathrm{U}, \mathrm{R}, \mathrm{W}, \wedge, \vee, \Rightarrow$.

E.g., $\mathrm{F}\, p \wedge \mathrm{G}\, q \vee \neg q \,\mathrm{U}\, p$ stands for $(((\mathrm{F}\, p) \wedge (\mathrm{G}\, q)) \vee ((\neg q)\,\mathrm{U}\, p))$.

Formal semantics LTL formulas are evaluated in the context of a transition structure $\mathcal{M}$. The satisfaction relation $\models$ is between paths in $\mathcal{M}$ and LTL propositions.

We call a state s of $\mathcal{M}$ *final* if it has no outgoing edges. In general, we focus on transition structures $\mathcal{M}$ without final states. This entails that every finite path in $\mathcal{M}$ can be extended to an infinite path. Such infinite paths represent possible full, completed histories of the world. The absence of final states embodies the idea that nothing can happen that stops time. This assumption is also made in the LTC, since there the time type `T` is interpreted by the natural numbers.

The limitation to transition structures without final states is not restrictive. A transition structure $\mathcal{M}$ with final states can easily be turned into an equivalent one $\mathcal{M}'$ without final states, by adding a dummy state s_d and extending $\rightarrow$ with edges to s_d from all final states and from s_d itself. Formally:

- $S' := S \cup \{s_d\}$
- $\rightarrow' := \rightarrow \cup \{(s_d, s_d), (s, s_d) \mid s \text{ is a final state of } \mathcal{M}\}$

The semantics of temporal logic is expressed in terms of *infinite paths* in $\mathcal{M}$. This is as in LTC, where time is also infinite.

Definition 4.2.2. A *path* in $\mathcal{M} = \langle S, \rightarrow, L\rangle$ is an infinite sequence of states $\langle s_0, s_1, s_2, \ldots\rangle$ such that $s_i \rightarrow s_{i+1}$ for $i \geq 0$.

We use the mathemical variable π to denote paths. Paths are often written informally as $s_0 \rightarrow s_1 \rightarrow \ldots$. The state s_i in such a path is called the state at time i in the path. The *suffix path* of a path π starting at s_i is denoted π^i. E.g., π^3 denotes the path $s_3 \rightarrow s_4 \rightarrow s_5 \rightarrow \ldots$. A special case is that $\pi^0 = \pi$. A path starting at state s will sometimes be denoted as $s \rightarrow \ldots$ or $s \rightarrow s_1 \rightarrow \ldots$.

Definition 4.2.3. Given $\mathcal{M} = \langle S, \to, L \rangle$ over Σ. Let $\pi = s_0 \to \dots$ be a path in $\mathcal{M}$ and φ an LTL formula over Σ.

We define that π satisfies φ (notation, $\mathcal{M}, \pi \models \varphi$) by induction on the structure of φ:

- $\mathcal{M}, \pi \models \top$ (and $\pi \not\models \bot$)
- $\mathcal{M}, \pi \models p$ if $p \in L(s_0)$
- the normal rules for logical connectives $\neg, \wedge, \vee, \Rightarrow$
  E.g., $\mathcal{M}, \pi \models \psi \Rightarrow \varphi$ if $\mathcal{M}, \pi \models \varphi$ or $\mathcal{M}, \pi \not\models \psi$
- $\mathcal{M}, \pi \models \mathrm{X}\,\varphi$ if $\mathcal{M}, \pi^1 \models \varphi$
- $\mathcal{M}, \pi \models \mathrm{G}\,\varphi$ if, for all $i \geq 0$, $\mathcal{M}, \pi^i \models \varphi$
- $\mathcal{M}, \pi \models \mathrm{F}\,\varphi$ if, for some $i \geq 0$, $\mathcal{M}, \pi^i \models \varphi$
- $\mathcal{M}, \pi \models \psi\,\mathrm{U}\,\varphi$ if there exists $i \geq 0$ such that $\mathcal{M}, \pi^i \models \varphi$ and for all $j = 0, \dots, i-1$, $\mathcal{M}, \pi^j \models \psi$
- $\mathcal{M}, \pi \models \psi\,\mathrm{W}\,\varphi$ (Weak until) if
  - either there exists $i \geq 0$ such that $\mathcal{M}, \pi^i \models \varphi$ and for all $j = 0, \dots, i-1$, $\mathcal{M}, \pi^j \models \psi$,
  - or for all $j \geq 0$, $\mathcal{M}, \pi^j \models \psi$
- $\mathcal{M}, \pi \models \psi\,\mathrm{R}\,\varphi$ (Release) if
  - either there exists $i \geq 0$ such that $\mathcal{M}, \pi^i \models \psi$ and for all $j = 0, \dots, i$, $\mathcal{M}, \pi^j \models \varphi$,
  - or for all $j \geq 0$, $\mathcal{M}, \pi^j \models \varphi$

If $\mathcal{M}$ is clear from the context, we write $\pi \models \varphi$ rather than $\mathcal{M}, \pi \models \varphi$.

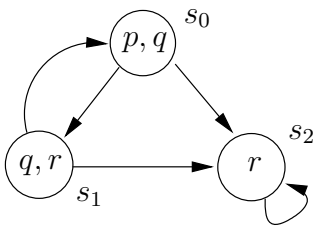
The place where $\mathcal{M}$ is used in the definition is at the base case: $\mathcal{M}, \pi \models p$ if $p \in L(s)$.

Now we define a second satisfaction relation between states and LTL formulas. An LTL formula is satisfied in state s if it is satisfied in each path from state s.

Let $\mathcal{M} = \langle S, \to, L \rangle$ and $s \in S$.

Definition 4.2.4. $\mathcal{M}, s \models \varphi$ if for each path $\pi = s \to \dots$ in $\mathcal{M}$ starting in s, it holds that $\mathcal{M}, \pi \models \varphi$

Example 4.2.2. What is true?



- $\mathcal{M}, s_0 \models p \wedge q$? true
- $\mathcal{M}, s_0 \models \neg r$? true
- $\mathcal{M}, s_0 \models \mathrm{X}\, r$? true
- $\mathcal{M}, s_0 \models \mathrm{G}\, \neg(p \wedge r)$? true
- $\mathcal{M}, s_0 \models \mathrm{F}\, \mathrm{G}\, r$? false; counterexample $(s_0 \to s_1 \to)^\infty$
- for all s, $\mathcal{M}, s \models \mathrm{F}(\neg q \wedge r) \Rightarrow \mathrm{F}\, \mathrm{G}\, r$? true
- $\mathcal{M}, s_0 \models \mathrm{F}\, \mathrm{G}\, r \Rightarrow \mathrm{G}\, \mathrm{F}\, r$? true

4.2.2 Translation to FO

LTL formulas can easily be translated into FO formulas of LTC. Given a propositional LTL-vocabulary Σ, we use the vocabulary Σ_{LTC} consisting of a unary predicate symbol $p(\mathtt{T})$ for every $p \in \Sigma$.

An LTL formula φ is translated into a FO-formula that expresses φ is true at some time term t. The corresponding formula is denoted φ^t. This formula is defined by induction:

- $p^t = p(t)$, where $p \in \Sigma$;
- $(\psi \wedge \phi)^t = \psi^t \wedge \phi^t$, and the same for $\vee, \Rightarrow, \neg$
- $(\mathrm{X}\, \varphi)^t = \varphi^{t+1}$
- $(\mathrm{G}\, \varphi)^t = \forall t_1 (t_1 \geq t \Rightarrow \varphi^{t_1})$
- $(\mathrm{F}\, \varphi)^t = \exists t_1 (t_1 \geq t \wedge \varphi^{t_1})$
- $(\psi\, \mathrm{U}\, \varphi)^t = \exists t_1 (t_1 \geq t \wedge \forall t_2 ((t \leq t_2 < t_1) \Rightarrow \psi^{t_2}) \wedge \varphi^{t_1})$

Exercise 4.2.1. *Extend the translation for* W, R.

Example 4.2.3. We apply the translation to compute:

$$(\mathrm{F}\, \mathrm{G}\, r \Rightarrow \mathrm{G}\, \mathrm{F}\, r)^0 = ?$$

The result is:

$$\exists t_1 (t_1 \geq 0 \wedge \forall t_2 (t_2 \geq t_1 \Rightarrow r(t_2))) \Rightarrow \forall t_1 (t_1 \geq 0 \Rightarrow \exists t_2 (t_2 \geq t_1 \wedge r(t_2)))$$

Recall that a path $\pi = s_0 \to \ldots$ corresponds to a linear time structure $\mathfrak{A}_\pi$ such that $L(s_i)$ corresponds to the state in $\mathfrak{A}_\pi$ at time i.

Theorem 4.2.1. $\pi \models \varphi$ *iff* $\mathfrak{A}_\pi \models \varphi^0$.

No proof.

We see that LTL formulas translate to FO formulas that range over the entire natural numbers. E.g., GFp. In general, these formulas cannot "safely" be evaluated in a finite interval of time points. So finite model generation does not work well. Special techniques were developed to reason efficiently about such formulas. LTL was developed from this work.

4.2.3 Practical patterns of specifications

- It is impossible to get to a state in which *started* holds and *ready* not: $\mathrm{G}\,\neg(\mathit{started} \wedge \neg \mathit{ready})$
- If a *request* of some resource occurs, it will eventually be *acknowledged*: $\mathrm{G}(\mathit{request} \Rightarrow \mathrm{F}\,\mathit{acknowledged})$
- A certain device is *enabled* infinitely often: $\mathrm{G}\,\mathrm{F}\,\mathit{enabled}$
- A process will run into a state of eternal *waiting*: $\mathrm{F}\,\mathrm{G}\,\mathit{waiting}$
- An upward traveling lift at second floor does not change direction when it has passengers for the 5th floor: $\mathrm{G}(\mathit{floor}2 \wedge \mathit{up} \wedge \mathit{ButtonPressed}5 \Rightarrow (\mathit{up}\,\mathrm{U}\,\mathit{floor}5))$

Propositions that cannot be expressed in LTL:

- Statements that there exists a path to some state, or equivalently, that it is *possible* to reach a certain state. E.g. in any state, it is *possible* to get to a *ready* state.
- More in general, statements expressing that there is a path satisfying certain properties. E.g. The lift *could* remain on the third floor with its doors closed forever.
- For this sort of properties, we need CTL.

4.2.4 Important equivalences of LTL

Definition 4.2.5. Two LTL formulas φ, ψ are equivalent (written $\varphi \equiv \psi$), if for all transition structures $\mathcal{M}$ and all paths π in $\mathcal{M}$, it holds that $\mathcal{M}, \pi \models \varphi$ iff $\mathcal{M}, \pi \models \psi$.

As a consequence, when $\varphi \equiv \psi$, it holds that $\mathcal{M}, s \models \varphi$ iff $\mathcal{M}, s \models \psi$.

Duality All connectives and operators have a *dual* connective or operator. E.g., $\wedge$ and $\vee$ are called *dual*, since $\neg(\psi \wedge \phi) \equiv \neg\psi \vee \neg\phi$, and $\neg(\psi \vee \phi) \equiv \neg\psi \wedge \neg\phi$. Similarly, temporal operators have a dual operator:

- $\neg\,\mathrm{X}\,\phi \equiv \mathrm{X}\,\neg\phi$: X is self-dual
- $\neg\,\mathrm{G}\,\phi \equiv \mathrm{F}\,\neg\phi \quad \neg\,\mathrm{F}\,\phi \equiv \mathrm{G}\,\neg\phi$
- $\neg(\phi\,\mathrm{U}\,\psi) \equiv \neg\phi\,\mathrm{R}\,\neg\psi \quad \neg(\phi\,\mathrm{R}\,\psi) \equiv \neg\phi\,\mathrm{U}\,\neg\psi$

Other useful equivalences are as follows:

- $\mathrm{F}\,\phi \equiv \top\,\mathrm{U}\,\phi \qquad \mathrm{G}\,\phi \equiv \bot\,\mathrm{R}\,\phi \qquad \mathrm{G}\,\phi \equiv \phi\,\mathrm{W}\,\bot$
- $\phi\,\mathrm{U}\,\psi \equiv \phi\,\mathrm{W}\,\psi \wedge \mathrm{F}\,\psi$
- $\phi\,\mathrm{W}\,\psi \equiv \phi\,\mathrm{U}\,\psi \vee \mathrm{G}\,\phi$
- $\phi\,\mathrm{W}\,\psi \equiv \psi\,\mathrm{R}(\psi \vee \phi)$
- $\phi\,\mathrm{R}\,\psi \equiv \psi\,\mathrm{W}(\phi \wedge \psi)$

Exercise 4.2.2. *Prove these equivalences using the definition of satisfaction.*

It follows from these equivalences that every temporal connective operator except X can be expressed in terms of U. In turn, U can be expressed in terms of W as well as in terms of R. It follows that all temporal connectives can be written in terms of those operators as well.

Exercise 4.2.3. *Express* $p\,\mathrm{W}\,q$ *in terms of* U.

Adequate sets of LTL connectives The above equivalences imply that not all connectives need to be implemented in an LTL model checking algorithm. Only a few operators need to be "implemented".

X is needed because it cannot be expressed by other connectives. Of the remaining temporal connectives, only one from $\{\mathrm{W}, \mathrm{U}, \mathrm{R}\}$ is required.

Each of the following sets is called an *adequate set of LTL connectives*:

$$\{\mathrm{U}, \mathrm{X}\}, \{\mathrm{R}, \mathrm{X}\}, \{\mathrm{W}, \mathrm{X}\}.$$

Each temporal formula can be translated in a formula using only the temporal connectives of an adequate set.

Exercise 4.2.4. *Verify that all temporal connectives can be expressed using* X *and* U.

4.2.5 Example: mutual exclusion

We apply the modelling methodology on a simple protocol to ensure that concurrent processes do not access a shared resource simultaneously. This is done by identifying *critical sections* of the code, and taking care that only one process can be in its critical section at the same time.

Processes apply to enter their critical section with the protocol. The protocol determines which process is allowed to enter.

Desired properties of the protocol:

- Only one process in its critical section at any time.
- If a process requests to enter , it will eventually be permitted.
- A process can always request to enter its critical section.
- The two processes do not need to alternate in entering their critical section. That is, it should be possible that one process enters its critical section two times in a row.

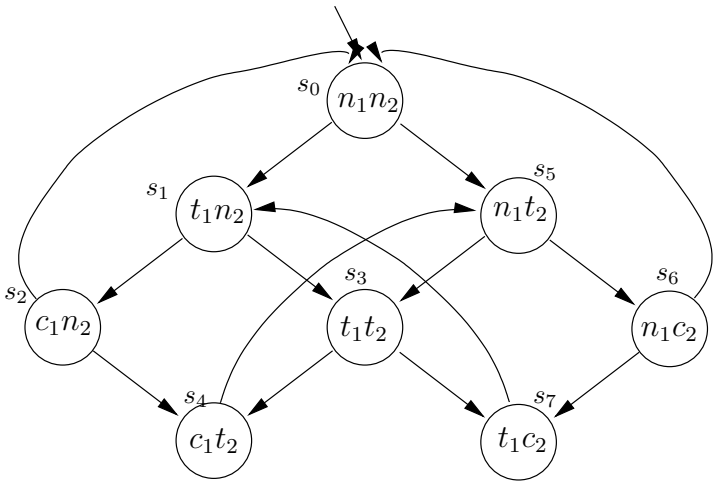


Figure 4.6: The mutual exclusion transition structure

The transition structure The protocol is expressed as a transition structure $\mathcal{M}$.

We assume that there are only two processes 1, 2.

- Vocabulary $\Sigma = \{n_i, t_i, c_i \mid i \in \{1, 2\}\}$. Each process i can be in one of three states:
  - n_i: normal,
  - t_i: applying for its critical section,
  - c_i: in its critical section.
- A state of the transition structure is made up by a pair $\{p_1, p_2\}$, with $p_1, p_2 \in \{n, t, c\}$.

We assume *asynchronous interleaving*: processes don't change state simultaneously.

The proposed transition structure is given in Figure 4.6.

Exercise 4.2.5. *Write an LTC theory to represent this application. Introduce types* $Process = \{1, 2\}$, $State = \{n, t, c\}$, *inertial fluent function symbol* $CurState(Process, \mathtt{T}) : State$ *and actions* $Request(Process, \mathtt{T})$, $EnterCriticalZone(Process, \mathtt{T})$ *and* $LeaveCriticalZone(Process, \mathtt{T})$.

Verifying desired propositions

- *Safety*: Only one process in its critical section at any time:

  $$\mathrm{G}\,\neg(c_1 \wedge c_2)$$

  Is it satisfied in s_0? yes.

- *Liveness*: If a process requests to enter the critical section, it will eventually be permitted:

  $$\mathrm{G}(t_i \Rightarrow \mathrm{F}\,c_i)$$

  Is it satisfied in s_0? No.

- *Non-blocking*: It is always possible for a process to request to enter its critical section. More precisely, each non-critical state n_i has a requesting next state t_i.
  A query for a *possible* path: expressible in CTL but not in LTL!
  Is it satisfied in s_0? Yes.

- *No strict sequencing*: processes need not enter their critical section in a strict alternation.

  We want to verify that there is a path with the property that "*it has two distinct states satisfying c_1 such that c_1 is false in at least one intermediate state and c_2 is false in all intermediate states*".

  In LTL, we cannot express this property directly, since there is no way to express *there is a path such that ...*. However, we can still encode it, namely by expressing the negation of the subproposition "it has two distinct states ...". The negation is "whenever c_1 is true, then c_1 remains true forever or until a state where c_1 remains false forever or until a state that c_2 is true". This is encoded by

  $$\mathrm{G}(c_1 \Rightarrow c_1 \,\mathrm{W}(\neg c_1 \wedge (\neg c_1 \,\mathrm{W}\, c_2)))$$

  There is a path from s_0 that does not satisfy this.
  Which one? $s_0 \to s_1 \to s_2 \to s_0 \to s_1 \to s_2 \ldots$

  Therefore $\mathcal{M}, s_0 \not\models \mathrm{G}(c_1 \Rightarrow c_1 \,\mathrm{W}(\neg c_1 \wedge \neg c_1 \,\mathrm{W}\, c_2))$. This entails the desired conclusion, namely that there is no strict sequencing. The falsifying path of this formula is the path we were looking for.

For the last proposition, we applied a coding trick. LTL does not offer the possibility to express properties of the kind "some path leaving from state s satisfies proposition ϕ". However, suppose that $\neg\phi$ can be expressed as an LTL formula ψ. Then $\mathcal{M}, s \not\models \psi$ holds exactly when there is a path leaving from s that satisfies $\neg\psi$, hence a path that satisfies ϕ.

This encoding trick exploits the fact that satisfaction in a state is not closed under negation: $\mathcal{M}, s \not\models \phi \not\equiv \mathcal{M}, s \models \neg\phi$.

- $\mathcal{M}, s \not\models \phi$ means that ϕ is false in some path from s
- $\mathcal{M}, s \models \neg\phi$ means ϕ is false in each path from s.

The trick does not work for all properties "there exists a path ..." The following property *"for each path leading to a state t_i, there is a path from that state to a state with c_i"* is not expressible using this trick because its negation includes "There is a path ..." which cannot be expressed in LTL.

We conclude that the protocol contains a bug since it does not ensure fairness.

A correct model To solve the Liveness problem, we may split the state t_1t_2 to remember which process applied first and give it priority over the other. A transition structure corrected for fairness is in Figure 4.7.

4.2.6 Mutual exclusion in ProB

A ProB theory expressing the mutual exclusion protocol is given in Figure 4.8. Some explanation follows:

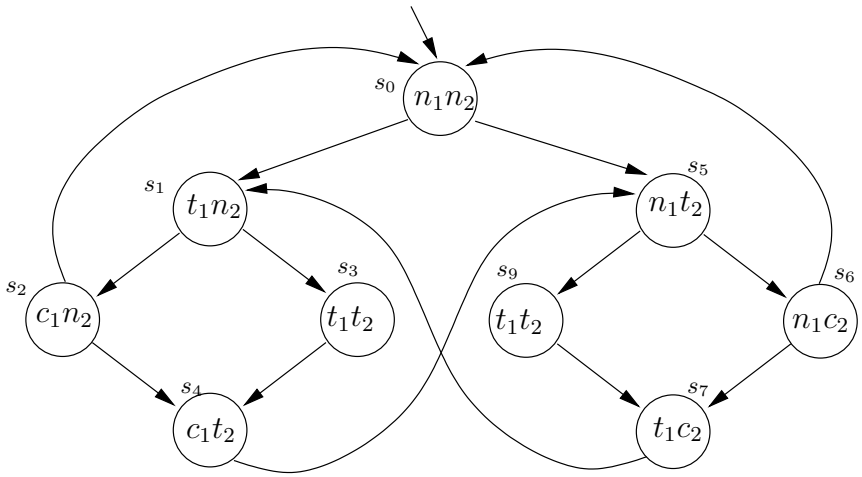


Figure 4.7: A fair mutual exclusion transition model

- `SETS: Proces, PrState`: this declares two set symbols and specifies their value. Their use is similar as type domains in IDP.
- `VARIABLES: Turn, State`: they are inertial fluent functions.
- `DEFINITIONS`:
  - `OtPr` is the static function mapping a process to another process.
  - Note the function notation `p1 |-> p 2`.
  - LTL statements are defined here: they do not belong to the modelling. Conditions in LTL are written between brackets, as in `GF({State(p1)=w})`. Observe that this is an extends propositional LTL.
- `INVARIANT:` here the types of fluents are declared together with other state formulas that should be invariant.
  - `Process -->Proces` is the set of total functions from Proces to Proces.
- `INITIALISATION`: the expression contains a concurrent assignment to `Turn` and `State`. The operator `||` is used to express concurrency, while `:=` is the standard assignment symbol.
- `OPERATIONS`: this describes preconditions in `PRE` and effects/assignments in `THEN` expressions.
- In the precondition of `Enter`, the expression `#x.(p(x))` is an existential quantification.

For more information, see exercise sessions and the following url's:

`http://www3.hhu.de/stups/prob/index.php?title=LTL_Model_Checking`

`http://www3.hhu.de/stups/prob/index.php/Mutual_Exclusion_(Fairness)`

```
MACHINE MutualExclusion
SETS
   Proces={p1, p2};
   PrState={n, w, c}
VARIABLES Turn, State
DEFINITIONS
   OtPr == {p1|->p2,p2|->p1};
   ASSERT_LTL == "GF({State(p1)=w})=> GF({State(p1)=c})";
   ASSERT_LTL1 == "GF({State(p2)=w})=> GF({State(p2)=c})"
INVARIANT
   OtPr:Proces-->Process &
   State:Proces-->PrState &
   Turn:Proces &
   not(State(p1)=c & State(p2)=c
INITIALISATION
   Turn:=p1 ||
   State:={ p1|->n , p2 |-> n }
OPERATIONS
Request(p) =
   PRE State(p)=n
   THEN State(p) := w
   END;
Enter(p) =
   PRE
      State(p)=t &
      not(#x.(State(x)=c)) &
      (State(OtPr(p))=w => Turn=p)
   THEN
      State(p) :=c ||
      Turn := OtPr(p)
   END;
Release(p) =
   PRE State(p)=c
   THEN State(p) := n
   END
END
```

Figure 4.8: Mutual exclusion in ProB

4.3 Fairness constraints

Example 4.3.1. A path on which c_2 is true *infinitely often* in Figure 4.7 is

$$s_0 \to s_1 \to (s_2 \to s_4 \to s_5 \to s_6 \to s_7 \to s_1 \to)^*$$

We say this path is fair with respect to c_2. A path on which c_2 is **not** true *infinitely often*:

$$s_0 \to s_5 \to s_6 \to s_0 \to (s_1 \to s_2 \to s_0 \to)^*$$

This path is "unfair" with respect to c_2.

Definition 4.3.1. A path π is fair with respect to some proposition φ if there are infinitely many $i \in \mathbb{N}$ such that $\pi^i \models \varphi$.

Theorem 4.3.1. *A path π of transtion structure $\mathcal{M}$ is fair with respect to φ iff $\mathcal{M}, \pi \models \mathrm{G}\,\mathrm{F}\,\varphi$.*

Exercise 4.3.1. *Prove this*

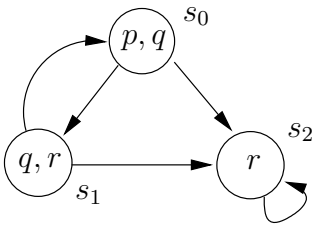
Many dynamic systems show natural fairness conditions.

- A process requests access to its critical section infinitely often.
- A process does not stay forever in its critical section. It means $\neg c_i$ is true infinitely often.
- For every floor, an elevator is called to it infinitely often.
- An elevator reaches all floors at regular times.
- Elevator requests are not ignored for eternity.

Fairness conditions, in general, cannot be expressed by a transition system. Therefore, some model checking systems offer the possibility to model the system by a combination of a transition structure $\mathcal{M}$ and a set C of fairness constraints.

Verification tasks of the system are relative to the set of paths of $\mathcal{M}$ that satisfy all fairness constraints $c \in C$.

Example 4.3.2. Assume the running example is extended with the fairness constraint $\neg q$.



The set of possible paths is $\{(s_0 \to s_1 \to)^n \to (s_2 \to)^\infty \mid n \in \mathbb{N}\}$.

This system entails $\mathrm{F}\,\mathrm{G}\,r$ while the original system did not.

Exercise 4.3.2. *How to express a fairness constraint φ in LTC? Can IDP reason on such constraints?*

4.4 CTL and CTL*: branching time logics

The motivation for Computation Tree Logic (CTL) is that propositions that express the existence of a path cannot be expressed in LTL. Sometimes, their negation can be expressed (the encoding trick!), but not in general. Yet, in many applications, existential statements of propositions are useful.

E.g., "from each reachable state in which P holds ($\forall$), we can reach a state in which Q holds ($\exists$)". This cannot be expressed in LTL. Neither can its negation "there is a reachable state in which P holds ($\exists$), from which Q is false on all paths ($\forall$)".

In branching-time logic, one can quantify over paths, both universally and existentially.

We see two branching time logics:

- CTL: *Computation Tree Logic*
- CTL*: a generalisation of both CTL and LTL

We begin with CTL*.

4.4.1 Syntax of CTL*

The idea of this logic is to distinguish between expressions that apply to states and expressions that apply to paths.

- *State formulas*: propositions about a state, hence to be evaluated in a state.
- *Path formulas*: propositions about a path, hence to be evaluated in a path.

Propositional formulas are about the current state, hence they are state formulas.

Temporal quantifiers $\mathrm{X}, \mathrm{F}, \mathrm{G}, \mathrm{U}, (\mathrm{W}, \mathrm{R})$ of LTL are evaluated in a path, hence they belong to path formulas.

Path quantifiers A, E quantify over paths starting from the current state. Hence, they belong to state formulas.

A state formula can be evaluated in a path, namely in its first state. Hence, a state formula is a special path formula. The inverse is not the case.

Definition 4.4.1. Syntax of CTL*.

We define *state formulas* and *path formulas* over vocabulary Σ by mutual induction using the following BNF:

- State formulas φ

$$\varphi ::= \begin{cases} \top \mid \bot \mid p \in \Sigma \mid \\ (\neg\varphi) \mid (\varphi \wedge \varphi) \mid (\varphi \vee \varphi) \mid (\varphi \Rightarrow \varphi) \mid \\ \mathrm{A}[\alpha] \mid \mathrm{E}[\alpha] \end{cases}$$

- Path formulas α

$$\alpha ::= \begin{cases} \varphi \\ (\neg\alpha) \mid (\alpha \wedge \alpha) \mid (\alpha \vee \alpha) \mid (\alpha \Rightarrow \alpha) \mid \\ (\mathrm{X}\,\alpha) \mid (\mathrm{F}\,\alpha) \mid (\mathrm{G}\,\alpha) \mid \\ (\alpha\,\mathrm{U}\,\alpha) \mid (\alpha\,\mathrm{W}\,\alpha) \mid (\alpha\,\mathrm{R}\,\alpha) \end{cases}$$

Notice the simultaneous induction of state and path formulas.

As stated before, a state formula is a path formula that refers to the first state of the path.

A path formula in CTL* is in general not a state formula. E.g., the path formula $\mathrm{F}\,\varphi$ expresses that eventually φ will become true. This is a property of a path. It is not a property of a state, since it may be true for one path leaving the state and false for another. However, both $\mathrm{A}\,\mathrm{F}\,\varphi$ and $\mathrm{E}\,\mathrm{F}\,\varphi$ are state formulas and can be evaluated in a state.

Binding priorities for CTL* are almost the same as for LTL

- First unary connectives: : $\neg, \mathrm{X}, \mathrm{F}, \mathrm{G}, \mathrm{A}, \mathrm{E}$;
- Then, in order: $\mathrm{U}, \mathrm{R}, \mathrm{W}, \wedge, \vee, \Rightarrow$

E.g., $\mathrm{A}[p \wedge q\,\mathrm{W}\,\neg p]$ is shorthand notation for $\mathrm{A}[p \wedge (q\,\mathrm{W}\,\neg p)]$.

4.4.2 Semantics of CTL*

Definition 4.4.2. Given $\mathcal{M} = \langle S, \to, L\rangle$.

We define the satisfaction relation for state formulas and path formulas by mutual induction on the structure of formulas:

- State formulas: for each state s and state formula φ, we define $\mathcal{M}, s \models \varphi$:
  - the standard rules for $\bot, \top$, atoms, $\neg, \wedge, \vee, \Rightarrow$;
  - $\mathcal{M}, s \models \mathrm{A}[\alpha]$ if for each path $\pi = s \to \ldots$ in $\mathcal{M}$, it holds that $\mathcal{M}, \pi \models \alpha$;
  - $\mathcal{M}, s \models \mathrm{E}[\alpha]$ if for some path $\pi = s \to \ldots$ in $\mathcal{M}$, it holds that $\mathcal{M}, \pi \models \alpha$;

- Path formulas: for each path $\pi = s_0 \to \dots$ and path formula α, we define $\mathcal{M}, \pi \models \alpha$:
  - the standard rules for $\neg, \wedge, \vee, \Rightarrow$,
  - the LTL rules for $\mathrm{X}, \mathrm{F}, \mathrm{G}, \mathrm{U}, \mathrm{W}, \mathrm{R}$.
  - if φ is a state formula: $\mathcal{M}, \pi \models \varphi$ if $\mathcal{M}, s_0 \models \varphi$.

Definition 4.4.3. Two path formulas are equivalent if they are satisfied in the same transition structures $\mathcal{M}$ and paths π. Two state formulas are equivalent if they are satisfied in the same transition structures $\mathcal{M}$ and states s.

Dualities and LTL-equivalences:

- $\mathrm{A}[\alpha] \equiv \neg\, \mathrm{E}[\neg\alpha]$
- $\mathrm{F}\,\alpha \equiv \neg\, \mathrm{G}\,\neg\alpha$
- $\alpha\, \mathrm{R}\, \beta \equiv \neg(\neg\alpha\, \mathrm{U}\, \neg\beta)$
- $\alpha\, \mathrm{W}\, \beta \equiv \alpha\, \mathrm{U}\, \beta \vee \mathrm{G}\,\alpha$

Note that R and W are "redundant".

4.4.3 Embedding LTL in CTL*

Syntactically, any CTL* path-formula without A, E is an LTL formula. An LTL formula α is evaluated in a path. The corresponding CTL* path formula is α itself.

An LTL formula α can also be evaluated in a state s according to the following rule: $\mathcal{M}, s \models \alpha$ iff for every path $\pi = s \to \dots$, $\pi \models \alpha$. This shows that the use of an LTL formula α as a state formula corresponds to the state formula $\mathrm{A}[\alpha]$.

Proposition 4.4.1. *$\mathcal{M}, s \models_{LTL} \alpha$ iff $\mathcal{M}, s \models_{CTL^*} \mathrm{A}[\alpha]$.*

When using LTL formulas as state formulas, the universal path quantifier is used implicitly. In CTL*, this quantifier is to be explicated.

We call $\mathrm{A}[\alpha]$ the embedding of the LTL formula α as a state formula.

4.5 CTL: a subformalism of CTL*

4.5.1 Syntax and semantics CTL

CTL-formula's are CTL* state formulas in which all temporal connectives come in pairs:

- $\mathrm{A\,X}$ and $\mathrm{E\,X}$
- $\mathrm{A\,F}$ and $\mathrm{E\,F}$
- $\mathrm{A\,G}$ and $\mathrm{E\,G}$
- $\mathrm{A}[\alpha\,\mathrm{U}\,\varphi]$ and $\mathrm{E}[\alpha\,\mathrm{U}\,\varphi]$

Definition 4.5.1. Syntax of CTL. We define the formulas of CTL by the BNF:

$$\varphi ::= \begin{cases} \bot \mid \top \mid p \mid \\ (\neg\varphi) \mid (\varphi \wedge \varphi) \mid (\varphi \vee \varphi) \mid (\varphi \Rightarrow \varphi) \mid \\ \mathrm{AX}\,\varphi \mid \mathrm{EX}\,\varphi \mid \mathrm{AF}\,\varphi \mid \mathrm{EF}\,\varphi \mid \mathrm{AG}\,\varphi \mid \mathrm{EG}\,\varphi \mid \\ \mathrm{A}[\varphi\,\mathrm{U}\,\varphi] \mid \mathrm{E}[\varphi\,\mathrm{U}\,\varphi] \end{cases}$$

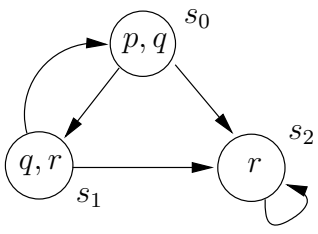
Each CTL-formula is a CTL*-*state formula.*

Informal semantics of CTL Expressions are evaluated in a some node s of the tree:

- $\mathrm{AX}\,\varphi$: for each path from s, φ holds in the next state;
  i.e., in all next states, φ holds
- $\mathrm{EX}\,\varphi$: in some path from s, φ holds in the next state;
  i.e., in some next state, φ holds
- $\mathrm{AF}\,\varphi$: each path goes through a state in which φ is true;
- $\mathrm{EF}\,\varphi$: some path goes through a state in which φ is true;
  i.e., φ is true in some state equal to or reachable from s;
- $\mathrm{AG}\,\varphi$: in each path, φ is true in all states;
  i.e., φ is true in all states equal to or reachable from s;
- $\mathrm{EG}\,\varphi$: in some path, φ is true in all states;
- $\mathrm{A}[\alpha\,\mathrm{U}\,\phi]$: in each path, $\alpha\,\mathrm{U}\,\phi$ is true;
- $\mathrm{E}[\alpha\,\mathrm{U}\,\phi]$: in some path, $\alpha\,\mathrm{U}\,\phi$ is true;

The meaning of some CTL expressions is illustrated in Figures 4.9 and 4.10.

Exercise 4.5.1. *Which formulas are true?*



- $M, s_0 \models \mathrm{EX}(q \wedge r)$?
- $M, s_0 \models \neg\,\mathrm{A}(p\,\mathrm{U}\,r)$?

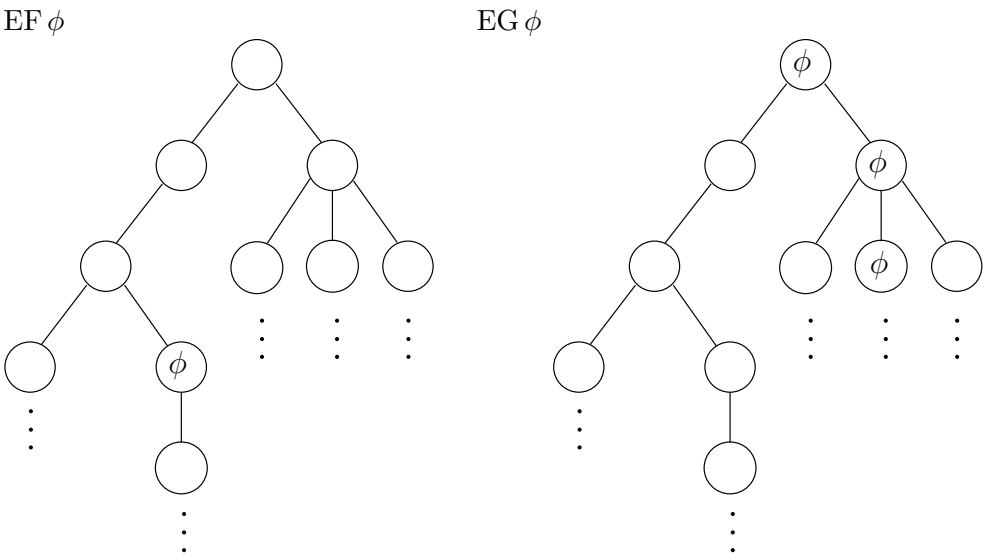


Figure 4.9: Truth of CTL formulas

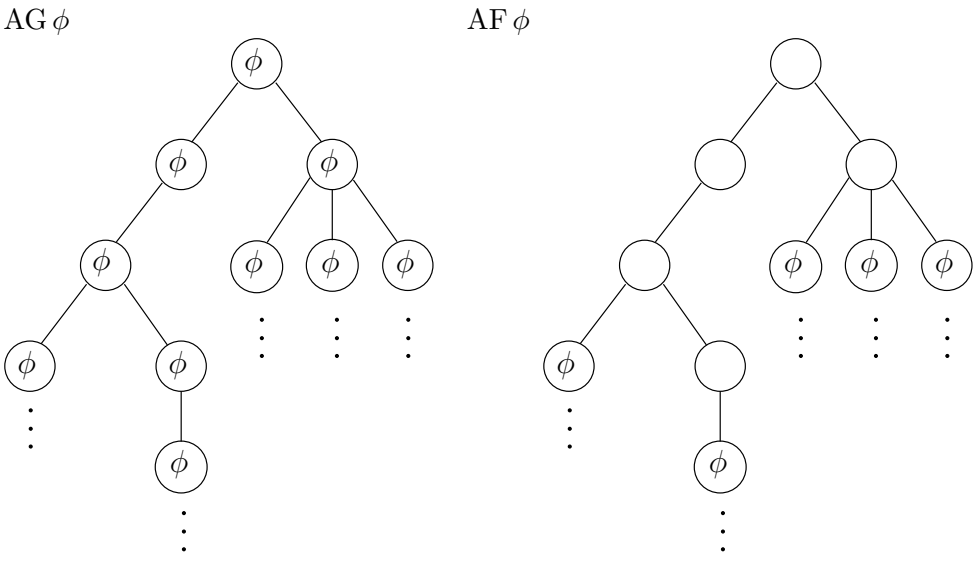


Figure 4.10: Truth of CTL formulas

- $M, s_0 \models \mathrm{E}[(p \wedge q)\,\mathrm{U}\,r]$?
- $M, s_0 \models \mathrm{EG}(q \wedge \mathrm{EG}\,\neg q)$?
- $M, s_0 \models \mathrm{EG}(q \wedge \mathrm{EX}\,\mathrm{AG}\,\neg q)$?
- $M, s_0 \models \mathrm{AG}(p \vee q \vee r \Rightarrow \mathrm{EF}\,\mathrm{EG}\,r)$?

4.5.2 Practical patterns of specifications

Assume that a transition system with an initial state is given. Below, a number of frequent informal state propositions are given, and how to translate them in CTL or LTL as state formulas of the initial state.

- All processes will eventually be deadlocked? This property is a universal quantification over all paths, and hence certainly expressible in LTL. It is expressible also in CTL (although there is a catch).
  - In CTL: $\mathrm{AF}\,\mathrm{AG}\,\mathit{deadlock}$
  - In LTL: $\mathrm{F}\,\mathrm{G}\,\mathit{deadlock}$

  Remarkably, these two statements are not logically equivalent. How can they be both correct then? For an explanation, see Remark 4.5.1.
- In its normal state, a process can always request to enter its critical section (using n_i, t_i, c_i)?
  - In CTL: $\mathrm{AG}(n_i \Rightarrow \mathrm{EX}\,t_i)$.
  - This property is not expressible in LTL. See Exercise 4.5.2.

  This is a statement about arbitrary states in which n_i holds. To express it in CTL, it is stated over all *reachable* states, using the operator AG. This is good enough in practice, but it does not exclude that there are states in the transition structure (unreachable from the initial state) in which the property does not hold.
- Processes need not enter critical section in strict alternating sequence?
  - In CTL: $\mathrm{EF}(c_1 \wedge \mathrm{EX}(\neg c_1 \wedge \mathrm{E}[\neg c_2\,\mathrm{U}\,c_1])) \vee \mathrm{EF}(c_2 \wedge \mathrm{EX}(\neg c_2 \wedge \mathrm{E}[\neg c_1\,\mathrm{U}\,c_2])))$.
  - Not in LTL. But as we saw, the negation of this proposition can be expressed in LTL: $\mathrm{G}(c_1 \Rightarrow (c_1\,\mathrm{W}(\neg c_1 \wedge \neg c_1\,\mathrm{W}\,c_2))) \wedge \mathrm{G}(c_2 \Rightarrow (c_2\,\mathrm{W}(\neg c_2 \wedge \neg c_2\,\mathrm{W}\,c_1)))$
- From any state it is possible to get to state *restart*:
  - $\mathrm{AG}\,\mathrm{EF}\,\mathit{restart}$
  - This property is not expressible in LTL. See theorem 4.5.1.
- When the elevator reaches the third floor with doors closed, it can stay there forever:
  - $\mathrm{AG}(\mathit{floor3} \wedge \mathit{doorsclosed} \Rightarrow \mathrm{EG}(\mathit{floor3} \wedge \mathit{doorsclosed}))$
  - Not in LTL. See theorem 4.5.1.
- Every process is enabled infinitely often.
  - In LTL: $\mathrm{G}\,\mathrm{F}\,\mathit{enabled}$

– In CTL: AG AF *enabled*.

Proof. First, assume G F *enabled* is true and AG AF *enabled* is false in the initial state. The latter means that EF EG ¬*enabled* is true. Or, there is a path to a state s having a path in which *enabled* is and remains false and in which F *enabled* is false. By concatenating both paths, we obtain a path from the initial state in which G F *enabled* is false. Contradiction.

Second, assume G F *enabled* is false and AG AF *enabled* is true. The first means that there is a path in which *enabled* is true at most a finite number of times. Take the state s on that path just following the last state where *enabled* is true. s is reachable from the initial state and AF *enabled* is false in s. Hence, AG AF *enabled* is false in the initial state. Contradiction.

- If a process is enabled infinitely often, then it runs infinitely often? (using *enabled*, *running*)
  - In LTL: G F *enabled* ⇒ G F *running*
  - In CTL? What about AG AF *enabled* ⇒ AG AF *running*? This is not correct!
    True, AG AF *enabled* is equivalent to G F *enabled*, and the same for AG AF *running* and G F *running*. Still,

    $$\text{G F } \textit{enabled} \Rightarrow \text{G F } \textit{running}$$

    and

    $$\text{AG AF } \textit{enabled} \Rightarrow \text{AG AF } \textit{running}$$

    are certainly not equivalent. The reason is the different quantification over paths. The LTL statement corresponds to the CTL* proposition

    $$A(\text{G F } \textit{enabled} \Rightarrow \text{G F } \textit{running})$$

    It quantifies once over every path from the initial state. The CTL expressions contains two separate quantifications over paths.

    To see the difference, assume there is one path where *enabled* is not infinitely often true. Then AG AF *enabled* is false and the CTL statement is trivially satisfied since its condition is false.

    But the second statement is trivally satisfied only for that one path. It still requires all other paths to satisfy G F *enabled* ⇒ G F *running*.

  This proposition is expressible in LTL and not in CTL.

Informal phrases containing "can" (as in "the elevator can stay on the 3rd floor") or "possible" (as in "it is possible to reach a state satisfying such or so") often refer to the existence of different potential futures. This is usually a sign that CTL is the appropriate language.

4.5.3 Logical analysis of CTL

Expressivity of LTL, CTL and CTL* Notice that CTL does not have path formulas that are not state formulas. Below, we compare the expressivity of CTL, CTL* and LTL on the level of *state formulas*.

Figure 4.11 shows a comparison of these logics.

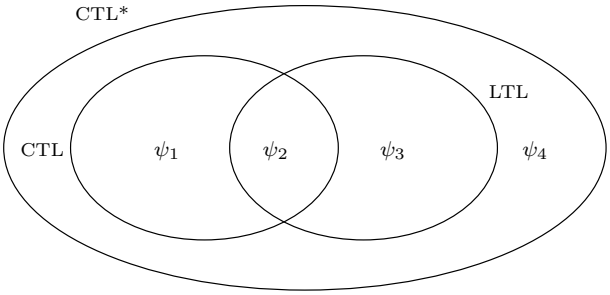


Figure 4.11: Expressivity comparison

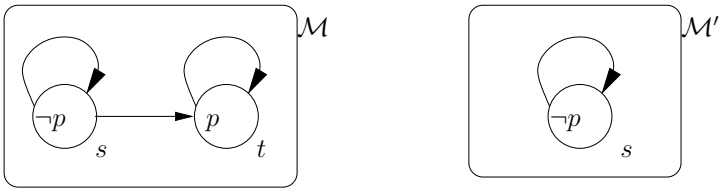


Figure 4.12: Structure with substructure for $\mathrm{AG}\,\mathrm{EG}\,p$.

- A formula $\psi_1 \in CTL \setminus LTL$: $\mathrm{AG}\,\mathrm{EF}\,p$. The proof is simple and illustrative.

  We call a transition structure $\mathcal{M}'$ a substructure of $\mathcal{M}$ if $\mathcal{M}'$ is obtained by deleting a set of edges from $\mathcal{M}$.

Theorem 4.5.1. . *Let φ be a CTL or CTL* formula such that there exists a transition structure $\mathcal{M}$ with substructure $\mathcal{M}'$ and a state s such that $\mathcal{M}, s \models \varphi$ and $\mathcal{M}', s \not\models \varphi$. Then φ is not expressible as an LTL formula.*

Proof. Assume towards contradiction that φ is expressible by LTL formula ψ. It follows that $\mathcal{M}, s \models_{LTL} \psi$ and $\mathcal{M}', s \not\models_{LTL} \psi$. But since $\mathcal{M}'$ is a subsystem of $\mathcal{M}$, each path of $\mathcal{M}'$ is still a path of $\mathcal{M}$ and hence, ψ is true on it. This contradicts $\mathcal{M}', s \not\models_{LTL} \psi$. ■

We illustrate the use of the theorem on the CTL proposition $\mathrm{AG}\,\mathrm{EG}\,p$. To show that it is not expressible in LTL, it suffices to find a structure and a state in which it is satisfied so that, after deleting some edges (and/or nodes), the proposition is not longer satisfied. In the structure $\mathcal{M}$ and substructure $\mathcal{M}'$ displayed in Figure 4.12, it holds that $\mathcal{M}, s \models \mathrm{AG}\,\mathrm{EF}\,p$ and $\mathcal{M}', s \not\models \mathrm{AG}\,\mathrm{EF}\,p$.

Exercise 4.5.2. *Prove that $\mathrm{AG}(n_i \Rightarrow \mathrm{EX}\,t_i)$ is not expressible in LTL. Prove the same for $\mathrm{EX}\,P \wedge \mathrm{EX}\,\neg P$. Think of other CTL formulas that cannot be expressed in LTL.*

- $\psi_2 \in CTL \cap LTL$:
  - in CTL: $\mathrm{AG}(p \Rightarrow \mathrm{AF}\,q)$
  - in LTL: $\mathrm{G}(p \Rightarrow \mathrm{F}\,q)$

Exercise 4.5.3. *Prove that these formulas are logically equivalent.*

A simpler example is $\mathrm{AG}\,\mathrm{AF}\,q$ and $\mathrm{G}\,\mathrm{F}\,q$.

- $\psi_3 \in LTL \setminus CTL$: $\mathrm{G}\,\mathrm{F}\,r \Rightarrow \mathrm{F}\,a$ (proof omitted)
  - It means that in all paths in which r is infinitely often true, then a will be satisfied. This may be a useful form of *fairness*: infinitely often requested (r) implies eventually acknowledged (a) at least one time. Notice that it does not require that a is true infinitely often.

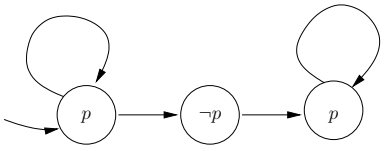
  Another example is $\mathrm{G}\,\mathrm{F}\,r \Rightarrow \mathrm{G}\,\mathrm{F}\,a$.

- $\psi_4 \in CTL^* \setminus (LTL \cup CTL)$: $\mathrm{E}[\mathrm{G}\,\mathrm{F}\,p]$ (proof omitted)
  - There is a path with infinitely many p's.

  Exercise 4.5.4. *Prove that the property cannot be expressed in LTL.*

From LTL to CTL? Above, we saw an LTL statement that can be translated into CTL by adding the path quantifier A in front of each LTL connective. E.g., from $\mathrm{G}(p \Rightarrow \mathrm{F}\,q)$ to $\mathrm{AG}(p \Rightarrow \mathrm{AF}\,q)$.

This is not always correct. The LTL statement $\mathrm{F}\,\mathrm{G}\,p$ is mapped in CTL: $\mathrm{AF}\,\mathrm{AG}\,p$. However, they are not equivalent. Here is a transition structure that makes one true and one false:



Exercise 4.5.5. *Which statement is true in the first state?*

As another example, the LTL formulas $\mathrm{X}\,\mathrm{F}\,p, \mathrm{F}\,\mathrm{X}\,p$ and CTL formula $\mathrm{AX}\,\mathrm{AF}\,p$ are all equivalent with each other but not with CTL formula $\mathrm{AF}\,\mathrm{AX}\,p$.

Exercise 4.5.6. *Explain the difference.*

Remark 4.5.1. On page 159, the proposition "all processes will eventually be deadlocked" was expressed

- in CTL as $\mathrm{AF}\,\mathrm{AG}\,\mathit{deadlock}$,
- in LTL as $\mathrm{F}\,\mathrm{G}\,\mathit{deadlock}$.

We just saw that these statements are not logically equivalent (taking $p = \mathit{deadlock}$). What is going on here? It is easy to see that $\mathrm{AF}\,\mathrm{AG}\,\mathit{deadlock}$ entails $\mathrm{F}\,\mathrm{G}\,\mathit{deadlock}$. So, the CTL formula is more restrictive. The transition system above is unnatural as a modelling of deadlocks, since it contains a deadlock state from which one can escape (but only to end up in another deadlock state in two steps). For transition systems that include deadlock in a "correct" way (namely as states from which there is no escape), the two formulas are equivalent.

Exercise 4.5.7. *Verify that in a transition structure where in every path on which deadlock becomes true, it remains true, the CTL and LTL formulas are equivalent.*

Useful equivalences of CTL

- $\mathrm{AX}\,\psi \equiv \neg\,\mathrm{EX}\,\neg\psi$
- $\mathrm{EG}\,\psi \equiv \neg\,\mathrm{AF}\,\neg\psi$
- $\mathrm{AG}\,\psi \equiv \neg\,\mathrm{EF}\,\neg\psi$
- $\mathrm{EF}\,\psi \equiv \mathrm{E}[\top\,\mathrm{U}\,\psi]$
- $\mathrm{A}[\psi\,\mathrm{U}\,\phi] \equiv \neg(\mathrm{E}[\neg\phi\,\mathrm{U}(\neg\psi \wedge \neg\phi)] \vee \mathrm{EG}\,\neg\phi)$

Adequate sets of CTL Below, paired connectives such as $\mathrm{EG}, AU, \ldots$ are considered as one CTL-connective.

Theorem 4.5.2. *A set of temporal connectives in CTL is adequate (i.e., suffices to express all CTL formulas) iff it contains one of* $\{\mathrm{AX}, \mathrm{EX}\}$ *, at least one of* $\{\mathrm{EG}, \mathrm{AF}, AU\}$ *and* EU*.*

Proof is omitted. (for if-part, the proof follows immediately from useful equivalences, for only-if part, the proof is more difficult.

On the usefulness of LTL and CTL and CTL*: Until 2000 20 years of debate about linear-time versus branching time logics with extensive literature comparing and arguing both.

LTL is to model the simplest and most common form of temporal propositions. Therefore, LTL is more often used in applications.

CTL allows to reason about different potential behaviours. An example:

$$\mathrm{EX}\,\mathrm{AG}\,p \wedge \mathrm{EX}\,\mathrm{AG}\,\neg p$$

This is a consistent CTL-statement about two potential and clearly incompatible behaviours of the system.

Exercise 4.5.8. *Give a transition structure and a state in which this CTL formula is true.*

To reason on potential behaviour is sometimes useful and cannot be simulated in LTL. But propositions about potential behaviour are inherently more complex and less frequently useful.

Why not CTL*? It's expressivity comes at a price: computationally much more expensive. LTL and CTL have model checking algorithms with linear time complexity in the size of the transition structure. Model checking algorithms for CTL* have higher complexity. In practical applications, transitions structures are very large which impedes the use of CTL*.

4.6 Important for exam

```
Big question for the theoretical part:
-  Define the logics LTL, CTL and CTL*:
formal definitions of syntax and semantics and discussion.
```

Small questions:
- prove the criterion for when CTL formulas cannot be expressed in LTL
- give formulas of LTL that cannot be expressed in CTL and vice versa
- explain some CTL or LTL formulas
- what is a fairness constraint
- evaluate a formula in a transition structure

What you need to know or understand to answer these questions:

Knowledge of
- transition structure
- LTL syntax and semantics
- CLT* syntax and semantics
- Integration of LTL in CTL*
- CTL as sublanguage of CTL*
- Expressivity limitations LTL/CTL: the criterion why many CTL statements cannot be expressed in LTL.

Chapter 5

Refinement in Event-B

5.1 Introduction

The goal of this course is to study modelling techniques and their applications. We study the use of modellings for verification and for solving software problems. We can use a formal modelling for solving a software problem by applying a suitable form of inference on our formal theory. Alternatively, we may sometimes be able to compile the formal modelling into a program that solves the software problem. In this chapter, we see a new key modelling technique: *Refinement*. It is a general technique to structure modellings of dynamic systems at different levels of abstraction; at the same time, it has also been used to build modellings that can be compiled into programs.

The modelling methodology seen so far aims to express knowledge directly in some target vocabulary Σ. However, for large systems, this method has its disadvantages. In particular, the vocabulary must be designed at the abstraction level suitable to solve the problems, and this abstraction level may be too refined, too concrete, too detailed. It may lead to potential problems in designing the vocabulary and correspondingly, problems with building the theory at this precision level. Moreover, proving correctness properties of this theory might be challenging.

The technique of step-wise refinement is based on the idea to build a specification in an iterative way, starting from a specification at a high abstract level and gradually refining it. An initial specification is step-wise refined until the desired level of concreteness is obtained. Each of these steps improves the specification in some way, by adding more detail: refining existing concepts or actions, adding new concepts and new actions that implement abstract concepts and actions and that add new functionality or lead to more efficient behaviour. During the refinement process, verification steps take place at all levels of abstraction. In some cases, the result is a specification that is concrete enough to be compiled to an executable program. [1]

An important aspect is that step-wise refinement is *incremental*: in the construction of the modelling and in the verification of properties. Refinement steps are constrained such that all verifications proven at higher level continue to hold. As such, the correctness of specifications at concrete levels depends and exploits verifications that were proven at higher abstraction levels. If all these refinements happen correctly the final specification (or program) must be correct by construction.

Compared to "one-shot" methodologies in which a specification is built in one step at the desired level of concreteness (as is currently the case in almost all declarative problem solving

paradigms), the refinement methodology has some important advantages. First, it leads to a more comprehensible, efficient, robust and flexible design process. Abstract specifications are simpler to understand; abstract specifications serve as documentation for more concrete ones. Bugs in a design are detected earlier, at an abstract level. Multiple alternative refinements can be explored, without losing the work done at higher abstraction level. Perhaps the most important advantage is for verification. In a "one-shot" methodology leading to a complex model, verification tasks may be of overwhelming complexity. In the refinement methodology, this complexity is broken up in small steps.

This way, refinement combines the advantages of modelling at an abstract level, with those of modelling at a concrete level. The advantages of modelling at a high abstraction level: ignoring irrelevant detail, focus on core functionality and structure, simplified reasoning. Abstraction is a key technique to exploit the power of mathematics in empirical formal sciences. Likely, abstraction will play a key role in mastering the complexity of software and other systems.

But modelling at an *abstract level* is insufficient. Abstract modellings do not contain sufficient information to solve your problem. By applying refinement, details of the problem domain can be brought in in order to *refine specifications from abstract to concrete.*

This chapter offers an overview of the refinement technique as it was developed for Event-B.

Historical note The step-wise refinement method was originally developed for programming by Dijkstra and Wirth. Today the approach is used for programs as well as for formal specifications. Step-wise refinement was originally proposed as a constructive approach to program proving. Starting from an initial simplified program, each refinement has to be carried out carefully so that it preserves the correctness of the program, while adding details. If all these refinements happen correctly the final program must be correct by construction.

This technique contrasts with other formal methods, where verification is used to prove program correctness. Using verification, correctness properties of the program or system can be proven *after* their development (e.g., Hoare logic). However, if any changes are made to the verified parts, the entire verification has to be repeated. When using refinement proving the correctness of the initial program as well as that of each refinement step is sufficient to prove the correctness of the final program. Verifying these steps is usually much simpler than the verification of an entire system. [2]

The B method constitutes a formal method which supports the use of refinement when modelling, verifying and developing systems. In particular the Event-B method, an evolution of B, is designed around refinement. Using the ideas of step-wise refinement, Event-B adopts a proof-based development strategy. For each refinement a number of proof obligations, i.e. conditions for correctness, are generated. These guarantee the correctness of the refinement step. Combined with the correctness of the initial modelling, this is enough to guarantee the correctness of the final modelling. These proof obligations are usually proved automatically. [3]

The B method and Event-B have been used to develop several safety-critical systems, including a driverless metro in Paris[1]. The next section will go into more detail about these examples. Section 5.3 will describe the refinement process.

[1] `https://en.wikipedia.org/wiki/Paris_M%C3%A9tro_Line_14`

5.2 Real world examples

The B method and Event-B are used to develop safety-critical systems. We discuss a few examples.

Metro 14 in Paris using the B-method On 15 October 1998 a new metro line opened in Paris. Metro line 14 is completely automated. Safety properties on several safety-critical parts of the systems were proved using the B method. To do so over 100 000 lines of B modelling were written. After proving no bugs were detected, neither at the functional validation, integration validation, on-site testing, nor since the metro started operating. The success of the B method is apparent when you look at the result. In 2009, the safety-critical software is still in version 1.0.

One aspect of the metro line, the operation of the platform screen doors, were almost completely verified using the B method. The system and software specifications were formalized in B and correctness was proved. Development of the doors' controller lasted four months, after which it was deployed on three platforms for an eight month experiment. No faults were observed during these eight months.

For more information, see `https://en.wikipedia.org/wiki/Paris_M%C3%A9tro_Line_14`

Other show cases More recently Event-B is being used for the development of safety-critical and other industrial projects. Event-B was designed as an alternative for B, specifically suited for modelling dynamic systems in their entirety. Event-B is being used by companies including:

Bosch Reliability and safety are essential for embedded control systems in the automotive industry. The required level of safety and dependability is achieved by means of testing. However due to the continuous increase in system complexity, the effort of testing will grow and will become uneconomical. To evaluate whether formal methods, and in particular Event-B, are suitable for the development and verification of automotive systems, Bosch applied the Event-B methodology to two applications, cruise control and (eco-)start/stop software.

Siemens Transportation Siemens has considerable experience in applying formal methods to software components of railway systems. They have successfully been using the B method for over 15 years. Between 2008 and 2012 Siemens looked into using Event-B for overall system development, applied to the development of transportation systems, results proved promising.

Huld Part of the BepiColombo space probe has been modeled using several levels of refinement. Event-B proved suited for the development of on-board software. The BepiColombo space probe is Europe's first mission to Mercury. It was launched on 20 oktober 2018.

The method of refinement for dynamic systems has not only been developed for the B-method. Even prior to the B-method, it was developed in the context of Abstract State Machines by Gurevitch starting from 1993. Event-B was developed by Abrial in 1996.

Using the refinement methodology, some spectacular successes have been obtained. However, so far it remains a costly design method which is only applied to mission critical.

5.3 Refinement

Refinement is a concept used in many fields and settings, going from topology refinement in mathematics to cultural refinement. In this course refinement is discussed in the context of formal methods, where it denotes transformations of abstract specifications into concrete systems, which preserve correctness and other properties. Step-wise refinement allows this transformation to be done in incremental stages.

While even in the context of formal methods the meaning refinement can vary, the basic idea of refinement is a simple one. It stems from the **principle of substitutivity**.

> If an item can be substituted by another, in such a way that its users can not tell a substitution has taken place, the latter is a refinement of the former.

A refinement B of a specification A has the property that at the abstraction level of A, the behaviour of the system specified by B is undistinguishable from the behaviour of the system specified by A.

Refinement is transitive in the sense that if B refines A and C refines B then C refines A. It follows that we can safely build hierarchies of refinements.

From the above intuition, in the context of dynamic systems, refinement can be defined as follows:

Definition 5.3.1. Refinement is the process of adding details to a modelling of a dynamic system, in such a way that the newly added information does not contradict what was already specified in the modelling. A refinement step links one *abstract* modelling to a more *concrete* modelling.

The goal when using refinement is to obtain a zooming effect on the system. Starting from an abstract, simple representation of the system details are added incrementally, to make the modelling more concrete. With each refinement we effectively zoom in on the system, allowing us to specify more details and describe the inner workings of the system. Compare this with a microscope; When you zoom in, you still see the same thing, but in more detail.

Example 5.3.1. To illustrate how refinement works and how it should be interpreted we describe a human using step-wise refinement. A first abstract description is as follows:

> A human is an animal.

This implies that humans have all general properties of animals. This entails that, like all animals, a human is a living organism, feeding on organic matter, having special sense organs and a nervous system and able to respond rapidly to stimuli.

We refine this description:

> A human is an animal *with the ability of walking.*

This refines the previous description with one additional detail. All properties entailed by the abstract description still hold. In addition, the new specification entails that humans are automotive and their way of movement is walking.

> A human is a *mammal* with the ability of walking.

Since mammals are animals, this refines the previous specification. It entails every proposition already entailed, but also that a human is warm-blooded, gives birth to live young and secretes milk for nourishment.

> A human is a primate which walks on two legs and uses language.

This brings us to a more accurate description of what a human is. If this description proves inadequate in a certain context, it can simply be further refined by adding additional, relevant details.

The example suggests that refinement is not only applicable for modelling dynamic systems but is a generally applicable methodology in knowledge representation. Although some initial research was done in the context of Artificial Intelligence, it has not been developed in a systematic way for KR. This is a promising research topic. In this course we will apply it for dynamic systems only.

Throughout this chapter refinement will be discussed in the context of the Event-B method. To support this the Event-B syntax and modelling approach is first described in the following subsection.

5.3.1 Event-B and Rodin

Event-B is based on the B method. It supports an event-driven approach for modelling dynamic systems. An Event-B model represents a system as a finite list of *state variables* which are modified by a finite amount of *events*. Events correspond to actions as seen in LTC. These events describe the possible behaviour of the system. *Invariants* specify properties that must be satisfied in every reachable state of the system, i.e., they must be satisfied in the initial state and maintained by the events.

Rodin is a development environment for Event-B modellings. It supports interactive Event-B modelling hierarchies by step-wise refinement and enforces a certain refinement strategy upon user. One important aspect is that it automatically generates *proof obligations* for the user.

A proof obligation is a proposition that must be proven to hold. E.g., adding a new event generates proof obligations for all existing invariants, namely that the event preserves the invariant. The proof obligations are generated automatically by Rodin. Proof obligations are verified semi-automatically by Rodin and may involve interaction with the user.

Modellings are split into static and dynamic parts. The static components are described in *contexts*. Contexts define constants and sets, these sets operate as types for the system. Constants are elements of the domain.

The dynamic part of the system is represented by *machines*. These consist of variables, invariants and events, as explained in the previous paragraph.

```
MACHINE
    SimpleMachine  ›
VARIABLES
◦   running  ›
INVARIANTS
◦   running_type:   running ∈ BOOL not theorem ›
EVENTS
◦   INITIALISATION:    not extended ordinary ›
    THEN
    ◦   off:    running ≔ FALSE ›
    END

◦   Start:  not extended ordinary ›
    WHERE
    ◦   off:    running = FALSE not theorem ›
    THEN
    ◦   start:  running ≔ TRUE ›
    END

◦   Stop:   not extended ordinary ›
    WHERE
    ◦   on: running = TRUE not theorem ›
    THEN
    ◦   stop:   running ≔ FALSE ›
    END

END
```

Figure 5.1: A simple Event-B machine

This approach makes for easily readable and understandable modellings of dynamic systems. Figure 5.1 illustrates this. The example shows a simple machine. The single state variable `running` is a boolean variable stating whether the machine is running or not. Two events can take place, the machine can be started or stopped.

Events are made up of *guards* and *actions*. The guards specify the preconditions of the event. Actions specify an effect of an event. They are specified as assignments to variables. The syntax used in Event-B is a combination of set-theoretical notations and first-order logic.

Notice that the notion of *action* is different in LTC and in Event-B.

5.3.2 Building a copier by refinement

We build a simplified modelling of a *copier*. We start from a very simple model: a machine that can be on or off. We will then specify extra functionalities of a copying machine by bringing detail in internal state and events.

Refinement 0 When modelling systems, it is common, and recommended, to start with an abstract representation of the system. The first, most abstract description of a copy machine is the simple machine represented in Figure 5.1. The state variable `running` is a boolean variable stating whether the machine is running or not. The two events are: the machine can be started or stopped. This is our starting point.

Events are made up of *guards* and *actions*. Guards following `WHERE` are the event's preconditions. Actions following `THEN` specify the effects of an event.

Invariants and guards can be annotated. E.g., `not theorem` means the proposition is not implied by other propositions (invariants or guards). For an axiom that is `theorem`, Rodin generates a proof obligation. Once this proof obligation is full-filled, Rodin knows that the truth of this

```
Select_Amount: not extended ordinary ›
ANY
◦  amount  ›
WHERE
◦  on: running = TRUE not theorem ›
◦  amount_type:  amount ∈ N1 not theorem ›
THEN
◦  set_goal:  copy_goal ≔ amount ›
END

Copy:  not extended ordinary ›
WHERE
◦  on: running = TRUE not theorem ›
◦  goal:  copy_goal > 0 not theorem ›
THEN
◦  end_copy:  copy_goal ≔ 0 ›
END
```

Figure 5.2: `Select_Amount` and `Copy` events of the simple copier

axiom follows from the used set of premises, and will not reactivate the proof obligation anymore as long as these premises exist in the modelling.

Events can be annotated. e.g., `extended` means that the event is a refinement of an event defined at a higher abstraction level. Guards and actions of the higher level are preserved for it. Events that are `not extended` are new events of the current refinement level.

Refinement 1: Adding amount selection and copy We add two events to select the amount of pages, and to perform the copying. The state of the machine is extended to express the number of pages:

- `copy_goal`: a new numerical variable

The new events:

- `Select_Amount`: sets `copy_goal` to a specified value
- `Copy`: resets `copy_goal` to zero.

The two original events are left unchanged. This is shown in Figure 5.2.

The refinement introduces a new state variable `copy_goal` in the `Variables` section (not displayed). The new events are `not extended` and have no abstract counterpart in the original machine. They are viewed as refinements of the null event `skip`. This means that like `skip`, they should not have an effect that can be observed at the abstract level (0). Therefore, they should only change *new* state variables (`copy_goal`). Otherwise, they would violate the *principle of substitutivity* as they would cause a state change at the abstract level without visible action.

In the refinement, we now want to assert a new invariant: the copier should be on if it has a goal:

$$copy\_goal > 0 \Rightarrow running = TRUE$$

For this new invariant, Rodin generates a *proof obligation* for each existing event, namely to prove that the event preserves the invariant.

Now, we observe a problem with the event `Stop`: it does not preserve the new invariant. Indeed, `Stop` sets `running` to false. When executed when `copy_goal>0` holds, `Stop` violates the invariant. Therefore, we need to refine this event.

For now, we interrupt the copier example, and introduce some more theoretical concepts that are needed to finish this example. After that, we will return to it.

5.3.3 Proof Obligations

Rodin automatically generates proof obligations for event-B modellings. These proof obligations are used to make sure the modelling has certain properties, e.g., that guards, actions and invariants are well defined and that invariants are preserved by the modelling. To show that the machine never violates the invariant, it must be proven that each event preserves the invariant. Thus for each combination of an event and an invariant a proof obligation is generated.

When refining a machine a number of proof obligations are generated to incrementally maintain a state where each event preserves each invariant. When adding a new event e, there are news proof obligations for each existing invariant i. Likewise, when adding a new invariant i, proof obligations are generated for each existing event e. These are called proof obligations for invariant preservation.

Since also `Initialisation` is also an event, it has a proof obligation for every invariant. Proving all these proof obligations ensures that the invariants hold in the initial state and are preserved by all events. Hence, the modelling of the system entails the specified invariants.

We define the proof obligation for invariant preservation.

Definition 5.3.2. A proof obligation for *invariant preservation* for an event e and invariant i takes the following form:

$$\forall s, s', x : A \wedge I(s) \wedge H_e(x, s) \wedge T_e(x, s, s') \Rightarrow I_i(s')$$

where

- A is the set of static axioms on constants and sets in the context associate with the machine;
- $I(s)$ is the conjunction of all invariants of the machine at state s;
- H_e is the guard of event e;
- $T_e(s, s')$ is the bistate formula expressing that state s' results from s by applying the actions of e;
- i is the invariant; $I_i(s')$ states that i is true in s'.
- x represents the event parameter(s).

This is the same idea as the verification method for proving invariants from bistate Linear Time Calculus theories.

Returning to our running example, we see that the abstract invariant $running \in BOOL$ is preserved by the abstract events as well as by the two new concrete events.

On the other hand, the concrete invariant $copy\_goal > 0 \Rightarrow running = TRUE$ is not by the abstract event `Stop`. We need to refine this event.

5.3.4 Event refinement

As shown by the problem with the `Stop` event, even when only new, independent information is added during a refinement it is seldom enough to only add new state variables and events. Changes to existing events are also often required.

In the running example, the abstract `Stop` event violates the concrete invariant. We do not want to change the original abstract machine. Therefore, we need to refine the `Stop` event in the refined machine.

Adding details to an abstract event during a refinement step is called *event refinement*. Event refinement is performed by two sorts of modifications: *guard strengthening* and *action simulation*. Both need to satisfy certain conditions to be correct. Rodin will generate these as proof obligations.

When the *guard strengthening* and *action simulation* conditions hold, the refinement e of an abstract event preserves all invariants i preserved by the abstract event. Thus, there is no need to generate proof obligations for invariant preservation of e and these invariants i.

Definition 5.3.3. The *guard strengthening condition* for an event e that is refined is the proposition expressed below that states that the concrete event can only be enabled if the abstract event is enabled. The formula is:

$$\forall x, s : A \wedge I(s) \wedge J(s) \wedge H_e(x, s) \Rightarrow G_e(x, s)$$

where

- A is the set of axioms in the context
- I and J represent the invariants of the abstract and concrete machine, respectively.
- H_e is the conjunction of all guards of the concrete event and G_e is the conjunction of the guards of the abstract event.

The guard strengthening condition expresses that in any reachable state where the concrete refined event can take place, also the abstract event can take place.

Definition 5.3.4. The *action simulation condition* for an abstract action a of an abstract event e that is being refined is the proposition that states that this action a "simulates" the concrete actions of the refined e. The formula is:

$$\forall x, s, s' : A \wedge I(s) \wedge J(s) \wedge H_e(x, s) \wedge T_e(x, s, s') \Rightarrow Q_a(x, s, s')$$

```
Start:  extended ordinary
REFINES
  Start
WHERE
  off:    running = FALSE not theorem
THEN
  start:  running ≔ TRUE
END
```

```
Start:  extended ordinary
REFINES
  Start
WHERE
  off:    running = FALSE not theorem
THEN
  start:  running ≔ TRUE
END
```

```
Stop:   extended ordinary
REFINES
  Stop
WHERE
  on: running = TRUE not theorem
  no_goal:    copy_goal = 0 not theorem
THEN
  stop:   running ≔ FALSE
END
```

```
Stop:   extended ordinary
REFINES
  Stop
WHERE
  on: running = TRUE not theorem
THEN
  stop:   running ≔ FALSE
  end_goal:   copy_goal ≔ 0
END
```

Figure 5.3: Two solutions for `Stop`

Where T_e is the bistate formula expressing the effects of the refinement of e and $Q_a(x, s, s')$ the bistate formula expressing the effect of the abstract action a.

T_e and is called the *before-after predicate* of the concrete event and Q_a that of the abstract action a.

This action simulation condition takes care that the event's abstract behaviour is not contradicted by its concrete behaviour.

Rodin generates a proof obligation for the guard strengthening condition of each refined event e and for the action simulation condition for each action a of the abstract event e.

5.3.5 Building a copier by refinement, continued

Refinement 1 rounded up The abstract event `Stop` violates the invariant

$$\texttt{copy\_goal} > 0 \Rightarrow \texttt{running} = \texttt{TRUE}$$

In Figure 5.3 two solutions are specified:

- to strengthen the guard of `Stop` by requesting `copy_goal = 0`; hence, a machine cannot be stopped until printing is finished;
- to add the action `copy_goal := 0` to `Stop`.

Both are possible. The first way is to refuse `Stop` if the specified `copy_amount` is non-zero. The second is that `Stop` sets this value to 0. The second seems the more reasonable solution since it makes it possible to stop the copier even when it has not finished printing.

Refinement 2 We introduce a second refinement: *pages are copied one by one, not all at the same time*. Hence, copying becomes a *process*.

```
CONTEXT
    Copier_State
SETS
    states
CONSTANTS
    off
    copying
    ready
AXIOMS
    states_partition:   partition(states, {off},
                        {copying}, {ready}) not theorem
END
```

Figure 5.4: Context specifying the possible states of the copier

Events do not represent processes. An important assumption made in Event-B is that events take no time, ie. they happen *instantaneously*. This means only one event can happen at a time and there is no concept of "during" an event. Specifically this means events can not represent processes.

The solution in Event-B is to simulate a process by introducing two or more events: a begin event for the process and an end event for it, and possibly intermediate state transition events of the process. In addition, a state variable to represent the progress of the process. A process may run for ever, in which case the end event will not occur.

We apply this technique to `Copy`. The `Copy` event is split up in three subevents: *begin, print page, end*.

In the refined machine, the copier can be in three different states: *off*, *on and in rest*, and *on and printing*. It follows that the variable `running` is no longer sufficient precise. A novel form of refinement is needed: *variable refinement*.

We refine the variable `running` by a new variable `state`, with possible values: `off`, `ready` and `copying`. These values are introduced in a *context*, as shown in Figure 5.4.

The context in Figure 5.4 declares set `states` and three constants. It also contains an axiom:

$$\texttt{partition(states,\{off\},\{copying\},\{ready\})}$$

This expresses that {{`off`},{`copying`},{`ready`}} is a *partition* of the set `states`: the union of the three sets is `states` and the intersection of the three sets is empty. The latter implies that `off` $\neq$ `copying`, `off` $\neq$ `ready`, `copying` $\neq$ `ready`. This is the Event-B way to express UNA+DCA for the type `states`.

Now, we refine the abstract variable `running`. The variable `running` has to be replaced by `state` in all invariants. Rodin enforces here a gradual strategy for this. This strategy does not immediately allow you to remove `running` from the concrete machine since this would contradict the abstract machine. Instead, the new, concrete variable `state` has to be *linked* to its abstraction `running` by a *linking invariant*. In this case, this is the formula:

$$running = TRUE \Leftrightarrow state = copying \lor state = ready$$

This defines the value of the abstract variable in terms of the concrete variable. It links the abstract machine with the concrete machine: the abstract copier is running iff the concrete copier is ready or copying.

Now that the abstract variable `running` is defined in terms of the concrete variable, Rodin allows to remove `running` in all concrete events and invariants and replace it by `state`.

```
Start_Copy:     not extended ordinary ›
WHERE
∘   ready:  state = ready not theorem ›
∘   goal:   copy_goal > 0 not theorem ›
THEN
∘   copy:   state ≔ copying ›
∘   copy_job:   copy_job ≔ copy_goal ›
END

Copy_Page:  not extended ordinary ›
WHERE
∘   copying:    state = copying not theorem ›
∘   copies_left:    copy_job > 0 not theorem ›
THEN
∘   copy:   copy_job ≔ copy_job − 1 ›
END

Finish_Copy:    not extended ordinary ›
REFINES
∘    Copy
WHERE
∘   ready:  state = copying not theorem ›
∘   done:   copy_job = 0 not theorem ›
THEN
∘   end_copy:   copy_goal ≔ 0 ›
∘   finish:  state ≔ ready ›
END
```

Figure 5.5: Events related to the copying process.

In the next step, the event `Copy` will be refined to represent the process of copying. For this, two new events are introduced `Start_Copy` and `Copy_Page`. These, respectively, signify the start of the copying process and the completion of one copy. In addition, the abstract `Copy` event is refined by the concrete `Finish_Copy` event. The two new events are not, and can not be, refinements of the `Copy` event, since after `Copy` the `copy_goal` is zero.

Since `Copy_Page` does not refine an abstract event it is not allowed to change the abstract variables, in particular the `copy_goal` variable. Still it should be possible to keep track of how many more copies are required to reach the goal. For this reason the `copy_job` variable is introduced, which stores the progress of the copying process. At the start of the copying process this variable will be set to the `copy_goal`, after which it will be decremented every time a copy is made. When this variable reaches zero, the copying process can be stopped by the `Finish_Copy` event. These events are listed in Figure 5.5.

To complete Refinement 2, additional invariants can/should be added for correct operation of the copier. They are presented in Figure 5.6. The first three invariants specify the types of the new variables and supply the link with the removed abstract variable. The last three invariants specify properties of the copier which should always be true, e.g., that the value for `copy_job` can not exceed that of `copy_goal`. These invariants are quite obvious, but as stated before it is good practice to specify them. Due to the mechanism of proof obligations, they help protect against modelling mistakes, e.g., incrementing `copy_job` rather than decrementing in the `Copy_Page` event.

All generated proof obligations succeed for this refinement step, proving the copier works as intended.

```
INVARIANTS
  state_type: state ∈ states not theorem ›
  linking_invariant:  running = TRUE ⇔ state = copying ∨ state = ready not theorem ›
  copy_job_type:  copy_job ∈ ℕ not theorem ›
  copy_job_max:   copy_job ≤ copy_goal not theorem ›
  copying_job:    copy_job > 0 ⇒ state = copying not theorem ›
  copying_goal:   state = copying ⇒ copy_goal > 0 not theorem ›
```

Figure 5.6: Specified invariants of the copier machine

5.3.6 Correctness of Refinement

Refinement builds a sequence $M_0, M_1, \ldots, M_n$ of machines, each M_{i+1} refining M_i. What is the desired correspondence between these machines?

Each machine specifies a collection of paths (linear time structures), collected in a state graph aka a transition structure on states. Two main desired correctness properties exist between the collections of paths of two theories M_i and its refinement M_{i+1}.

Definition 5.3.5. Correctness property 1: each path/linear time model of M_{i+1} abstracts into a path/linear time model of M_i.

This property ensures that each behaviour at the concrete level $i + 1$ matches a behaviour at the level i.

The different steps of event and variable refinement that were introduced so far, with guard strengthening and action simulation conditions, suffice to prove correctness property 1.

Theorem 5.3.1. *Let $M_0, M_1, \ldots, M_n$ be a refinement sequence such that all proof obligations were satisfied, then all invariants at all levels are entailed .*

Definition 5.3.6. Correctness property 2: each path/linear time model of M_i is the abstraction of a path/linear time model of M_{i+1}.

This property ensures that each behaviour at the abstract level remains possible at the concrete level.

This property is not entailed by the refinements steps that we saw. Hence, not every behaviour of the abstract machine is possible at the refined machine. Some behaviors get "lost". Sometimes this is desirable. Other times it is an indication of a bug. In a later section, we will see other proof obligations that will entail Correctness property 2.

Example 5.3.2. Assume that we make a mistake in Refinement 2 and define:

```
Copy_Page
WHERE
...
THEN
copy_job := copy_job-0
END
```

The concrete machine loops. A run of the abstract machine where `Copy` happens, cannot be simulated, since `Finish_Copy` can never be executed. Hence, Correctness property 2 does not hold.

On the other hand, Correctness property 1 holds. Each run of the refined machine is still simulated at the abstract level. E.g., a run with an infinite number of `Copy_Page` events correspond to a run with infinite number of `skip` events.

The above example shows a problem of *divergence*: an infinite loop of new events. Alternatively, it is also possible that due to guard strengthening, refined events are not allowed to take place when their abstract versions can take place.

Additional correctness properties need to be imposed to obtain Correctness property 2. This will be seen in a later section.

5.4 More about proof obligations (not for exam)

We introduce a new running example. Given is a function $f : [1, n] \to [0, M]$ specified in some Event-B context. We build a machine to compute the maximum of the range of f, i.e., the set $\{f(i) \mid i \in [1, n]\}$.

Refinement 0

```
Variable m
Invariant
   Inv0_1:  m∈ ℕ
Events
   INITIALISATION
   THEN m:=0
   END
   Maximum
   THEN m:=max(ran(f))
   END
```

Here, `ran(f)` denotes the range of f.

The proof obligations at level 0 are:

- Invariance of `Inv0_1:  m∈ ℕ`. We need to prove that `INITIALISATION` establishes it and `Maximum` preserves it.
- Well-definedness of expression `max(ran(f))` in the action `m := max(ran(f))` of the event `Maximum`. `max` is only defined on non-empty sets. Therefore, we need to prove that

$\texttt{ran(f)} \neq \emptyset$

Rodin generates these three proof obligations and tries to prove them, possibly with assistance of the user.

Refinement 1 Now, we refine the abstract machine. First we, define its invariants:

```
Variable m p q
Invariant
   Inv0_1:  m∈ ℕ
   Inv1_1: p∈ 1..n
   Inv1_2: q∈ 1..n
   Inv1_3: max(ran(f)) ∈ f([p,q])
Events
   ...
```

We use a naming convention to specify the level of invariants. E.g., `Inv0_1` is at level 0, `Inv1_1` at level 1. One invariant is of refinement level `(0)`, three are of level `(1)`.

Next, we define the refinements and new events.

```
INITIALISATION (0)
   THEN m:=0
        p:=1; q:= n
   END;
```

```
Maximum (0)
   WHEN p=q
   THEN m:=f(p)
   END;
```

```
increment (1)
   WHEN
      p<q
      f(p) ≤ f(q)
   THEN
      p:=p+1
   END;
```

```
decrement (1)
   WHEN
      p<q
      f(p) > f(q)
   THEN
      q:=q-1
   END;
```

`(0), (1)` are the levels where the events were introduced.

An example simulation of this system is given in Figure 5.7. The matrix below denotes the sequence of states with at each state the action on the right. The colored cells are those to which `p` and `q` point at: `p` points at the left one and `q` at the right one. The simulation ends with $m = 10, p = q = 5$.

The concrete events at level (1) and their abstract events at level (0) is depicted as follows (using abbreviations for event names):

$$\begin{array}{ccccccc} INIT & skip & skip & skip & skip & skip & Max \\ \uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow \\ INIT & Inc & Inc & Dec & Inc & Inc & Max \end{array}$$

The new actions refine skip.

This refinement generates several proof obligations at level (1):

1)INIT	7	3	10	8	10	9	$p = 1; q = 6; m = 0$
2)Increment	7	3	10	8	10	9	$p = 2; q = 6; m = 0$
3)Increment	7	3	10	8	10	9	$p = 3; q = 6; m = 0$
4)Decrement	7	3	10	8	10	9	$p = 3; q = 5; m = 0$
5)Increment	7	3	10	8	10	9	$p = 4; q = 5; m = 0$
6)Increment	7	3	10	8	10	9	$p = 5; q = 5; m = 0$
7)Maximum	7	3	10	8	10	9	$p = 5; q = 5; m = 5$

Figure 5.7: A simulation of the machine

- 3×4 invariant preservation proof obligations. There are three new invariants `Inv1_`$\{1, 2, 3\}$ of level `1` and 4 events.
- Event refinement proof obligations. Two events were refined, and for both each, a guard strengthening and action simulation conditions proof obligations arise.
- The well-definedness proof obligation for `max(ran(f))`.

This suffices to prove the invariants of level (0) and (1). It suffices to prove the Correctness property 1 but not the Correctness property 2.

Pathologies for Correctness 2 A potential problem with refining an event is that the refined event may never be executable. Indeed, there is no guarantee that the guard of the refined event will ever become true. Two sorts of problems may arise.

- *Early deadlock*: the guard of the new events could become false without reaching a state in which the refined guard of refined event becomes true. In that case, the refined event cannot be executed. In the running example this would lead to a sequence of increment and decrement actions leading to a state where their guards and that of `Maximum` are all false.

$$INIT \quad Inc \quad Inc \quad Dec \qquad\qquad (Max, Inc, Dec\ impossible)$$

- *Divergence*: an infinite sequence of new events takes place without ever reaching a state where the refined event applies.

$$INIT \quad Inc \quad Inc \quad Dec \quad \ldots \quad \ldots \quad \ldots \quad (Max\ never\ possible)$$

To avoid these pathologies, additional proof obligations need to be imposed for Correctness property 2. They are called *trace refinement* proof obligations:

To prevent early deadlock, a new proof obligation is added:

Definition 5.4.1. The first trace refinement condition is the disjunction of the guards of new and refined events.

Hence, when no new event can take place, a refined event can take place. This prevents early deadlock.

To prevent divergence, it must be proven that new events are *convergent*. The user proposes a *variant* : a numerical expression `v` that must be proven to be a positive natural number and that strictly decreases in value when a new event takes place.

In the example, a suitable variant is `q-p`.

Two new sorts of proof obligations are introduced:

Definition 5.4.2. The second trace refinement condition is : `v`$\in \mathbb{N}$ is invariant.

The third trace refinement condition is: `v` decreases in value when a *new event* takes place.

Hence, an infinite sequence of new events would lead to an infinite descending sequence of values for `v`. This is impossible due to the invariant `v`$\in \mathbb{N}$.

The three trace refinement conditions ensures that divergent sequences are impossible. After a finite sequence of new events, a refined event takes place.

A suitable *variant* in the running example is the numerical term `q-p`. This variant consists of the following components:

- The invariant $q - p \in \mathbb{N}$ which must be added. To prove it, a new invariant $p \leq q$ is introduced. This and the existing invariants imply invariant preservation of $q - p \in \mathbb{N}$.
- A before-after predicate expressing that each new event `Increment`, `Decrement` decreases $q - p$ need to be proven.

5.5 Compiling Event-B to programs (not for exam)

Event-B specifications that are sufficiently "concrete" can be compiled to a program. This section illustrates the essential steps in the context of the running example.

Pidgin Programming Language A simplified programming language with 4 sorts of statements:

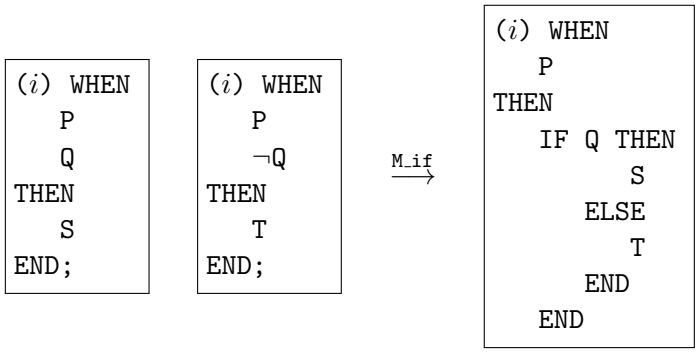
- *WHILE* condition *DO* statement *END*
- *IF* condition *THEN* statement *ELSE* statement *END*
- statement *;* statement
- variable list *:=* expression list

The assignment expresses concurrent assignments. E.g. *a, b := a+1, a* turns $a = 1, b = 0$ into $a = 2, b = 1$. This is not equivalent with *a := a+1; b := a*.

Merging rules Merging rules transform sets of events in an equivalent set of events of fewer but more complex events.

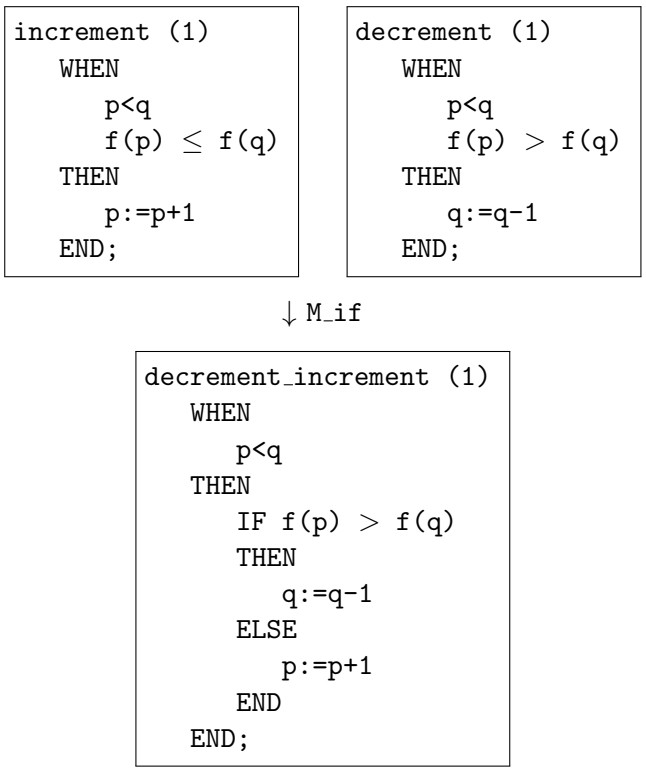
By iterated application of merging rules, an Event-B specification is gradually transformed in a Pidgin program.

Merging rule M_if Two events with mutually exclusive guards are combined in one:

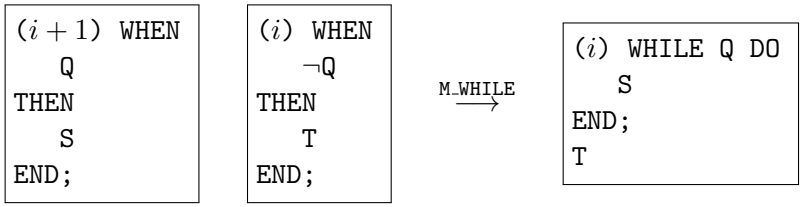


The two original events must have been introduced at the same refinement level i.

We apply this to the increment and decrement events of the previous section:

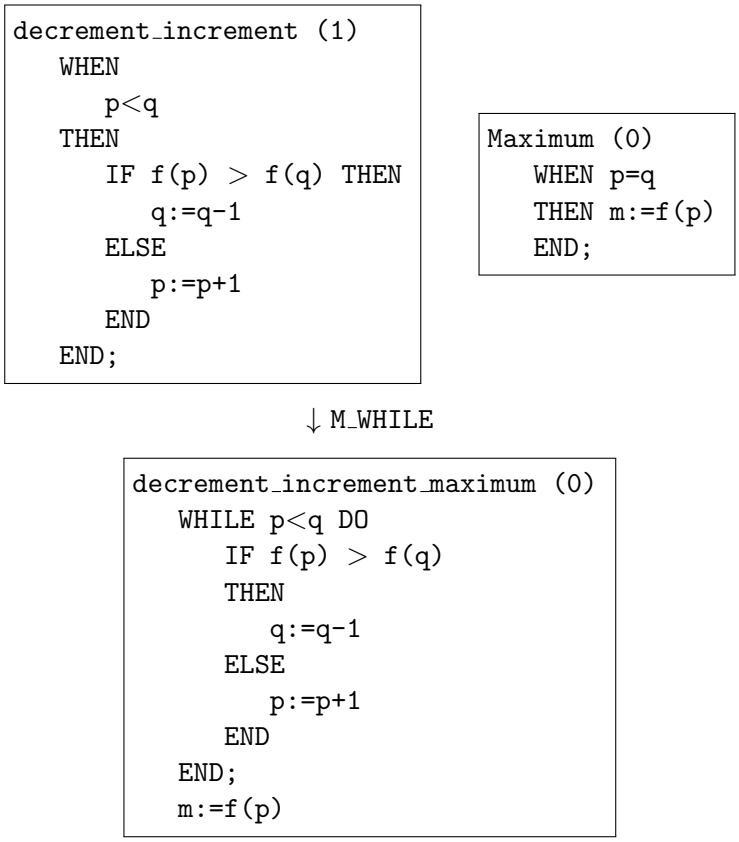


Merging rule *M_WHILE* Two events with mutually exclusive guards are combined in one:



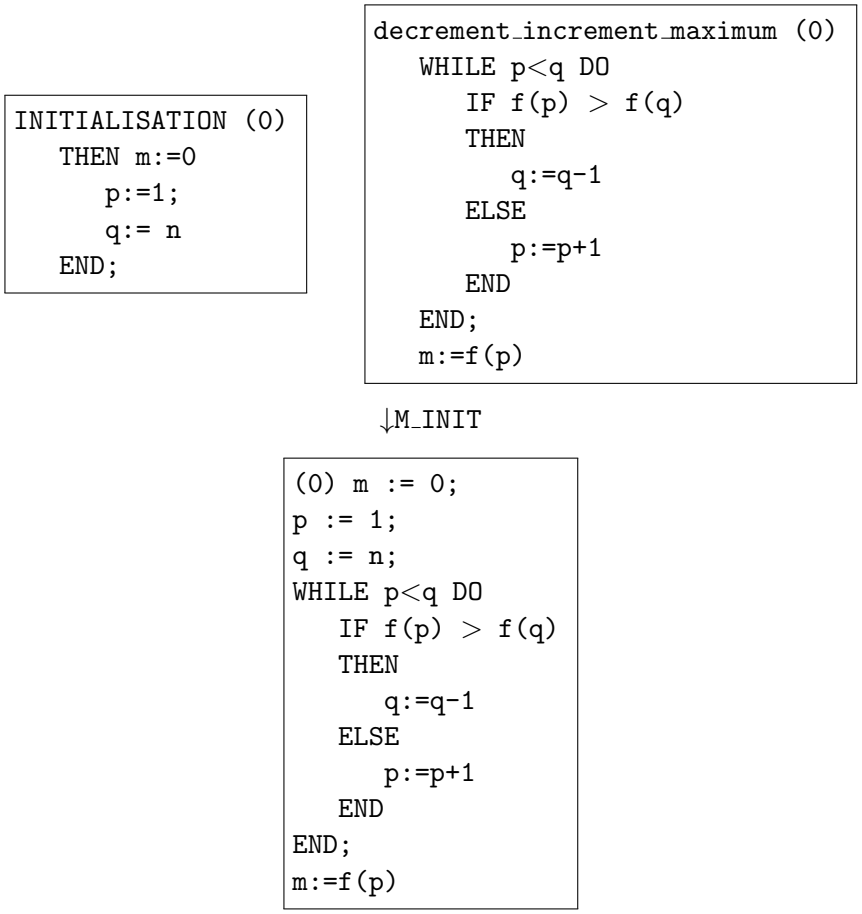
Condition: the first event must have been introduced at one refinement step below the second one. More general cases of this rule exist.

Applied to the example of the previous section:



Final rule *M_INIT* Once we have obtained an "event" without guard, we add it to the event `INITIALISATION`. We then obtain the final program.

Applied to the running example:



5.6 Conclusion

Refinement allows a modelling to be built gradually, from abstract to concrete, through a number of refinement steps.

It has been shown that if these refinement steps happen correctly, the final modelling will satisfy all invariants of the full model at all abstraction levels..

Refinement is an instance of a general, important, useful idea that is applicable not only in dynamic systems but in all systems. The idea is to build precise and detailed concrete specifications by a sequence of concretisation steps.

So far in this course, descriptions of dynamic systems were on an abstract level (with exception of LogicBlox theories, which specify concrete executable processes). Abstraction is good. But to be practically useful, we need a way to link abstract descriptions to more concrete "executable" ones. Refinement is a missing link as it allows to close the gap between abstract specifications and concrete ones.

5.7 Important for the exam

```
There will be no big questions on this chapter ( your understanding
of the principles were tested  in the project).
```

There might be small questions: e.g., explanation of the terms of this chapter in the context of an event-B example.

There will be no questions of Section 5, 6 of this chapter.

What you need to know to be able to answer such questions:

Understanding of
- the principle of refinement
- proof obligations
- what are the correctness conditions.

Chapter 6

Classical results on Predicate Logic Provability, Expressivity, Decidability of deduction, Incompleteness

6.1 Deductive inference

The following are deductive inference problems:

- The problem of deciding logic validity of a formula:

  Input: a formula φ
  Output: **t** if φ is logically valid ($\models \varphi$), and **f** otherwise

- The problem of deciding that a theory is unsatisfiable:

  Input: a theory T
  Output: **t** if T is unsatisfiable, **f** otherwise

- The problem of deciding logical consequence:

  Input: a theory T, formula φ
  Output: **t** if T logically entails φ ($T \models \varphi$), and **f** otherwise

For finite theories T these problems can be reduced to each other. E.g., φ is logically valid iff $\{\neg\varphi\}$ is unsatisfiable; T is unsatisfiable iff $T \models \mathbf{f}$; and $T \models \varphi$ iff $\wedge(T) \Rightarrow \varphi$ is logically valid. Here, ($\wedge(T)$ denotes the conjunction of the axioms of T. This is well-defined only for finite T.

A deductive inference algorithm is an algorithm to solve deductive inference problems. Deductive inference (or deduction for short) is the typical form of reasoning used in mathematical proofs.

Sometimes, deduction inference is called *truth preserving inference*. Indeed, from a set of properties that are true in the application domain it derives other properties that can be guaranteed to be true in that domain.

It was Aristoteles (350BC) who grew aware that there are syntactical patterns in such mathematical reasoning. He was the first to study them in a scientific way. He developed the theory of Aristotelian syllogisms. These are inference rules of natural language. An example of a syllogism is the following:

All greeks are mortal
Socrates is a greek

Socrates is mortal

For about 2000 year, his work set the standard. Philosophers and mathematicians have been searching since then to further formalize these patterns. Only in the 19th century substantial progress over Aristoteles was made. Leibniz, Boole, De Morgan, Gottlob Frege – the latter is considered to be the inventor of classical logic because he introduced quantification in the language.

The historical view on FO Classical logic as a modelling language was a by-product of the study of deductive inference. Indeed, any form of *reasoning* requires a crucial resource: *Information*. Information is the raw material to be fed into reasoning methods. To formally study reasoning, one needs a formal representation of information. Hence, one needs a formal language to express information. Leibniz, De Morgan, Boole, Frege not only contributed to deductive reasoning but also to the development of a formalism to express information. This resulted in the language of classical predicate logic.

A Hilbert proof system

Definition 6.1.1. A proof system consists of a set of *logical axioms* and *inference rules*.

The logical axioms are specified by *axiom schemata*: abstract formulas containing formula variables. They represent the set of all formulas that can be obtained by substituting formula variables by formulas.

E.g., a well-known schemata is $\alpha \vee \neg\alpha$. Here, α is the formula variable. A logical axiom is any instance of a schemata obtained by substituting the formula variables by concrete formulas. E.g., instances of the above schemata are $P \vee \neg P$, $Q(x) \vee \neg Q(x)$.

The inference rules $\frac{\beta_1,\ldots,\beta_n}{\alpha}$ specify that the formula α can be inferred from the formulas $\beta_1, \ldots, \beta_n$. That is, if we have proof of $\beta_1, \ldots, \beta_n$, application of the inference rule extends the proof with α.

One proof system of FO consists of the set of axiom schemata corresponding to propositional tautologies and the following two:

$$\forall x \alpha[x] \Rightarrow \alpha[t] \qquad \alpha[t] \Rightarrow \exists x \alpha[x]$$

where α is a formula in which x occurs free, t a term such that all its free symbols are free in $\alpha[x]$. It has three inference rules:

$$\frac{\alpha \Rightarrow \beta, \alpha}{\beta} \qquad \frac{\beta \Rightarrow \alpha[x]}{\beta \Rightarrow \forall x \alpha[x]} \qquad \frac{\alpha[x] \Rightarrow \beta}{\exists x \alpha[x] \Rightarrow \beta}$$

where $\alpha[x]$ is a formula with free variable x that does not occur free in β. The first inference rule is the modus ponens.

Another well-known proof system is for clausal logic: the resolution proof system which has no axiom schemata and one inference rule, the resolution inference rule.

Definition 6.1.2. A proof of ϕ from theory T in a given proof system is a finite sequence $\langle \alpha_0, \alpha_1, \ldots, \alpha_n \rangle$ such that $\alpha_n = \phi$ and each α_i

- is a logical axiom,
- or $\alpha_i \in T$ (we say that $\alpha_i \in T$ is *introduced* in the proof)
- or there is an inference rule $\frac{\beta_1, \ldots \beta_m}{\alpha_i}$ such that $\{\beta_1, \ldots \beta_m\} \subseteq \{\alpha_0, \alpha_1, \ldots, \alpha_{i-1}\}$. I.e., α_i is obtained by applying this inference rule on a set of propositions that were proven already.

There is seemingly an immense difference between verifying that a formula ϕ is true in all models of T, and building a proof of ϕ from T. Yet, a proof of ϕ from T is a "mechanical" way to establish that ϕ is true in all models of T, i.e., that $T \models \phi$.

Definition 6.1.3. Given is a proof system.

- A formula φ is *derivable* or *provable* from a FO theory T (denoted $T \vdash \varphi$) if there is a proof of φ from T.
- A FO theory T is *consistent* if there is no sentence φ, such that $T \vdash \varphi$ and $T \vdash \neg\varphi$.

The connection between provability and logical entailment, and consistence and satisfiability is laid in the concepts of soundness and completeness of a proof system.

Definition 6.1.4.

- A proof theory is *sound* if $T \vdash \alpha$ implies $T \models \alpha$.
- A proof theory is *complete* if $T \models \alpha$ implies $T \vdash \alpha$.

If a proof system is sound and complete, then $\vdash$ and $\models$ coincide. I.e., derivability and logical entailment coincide. But also consistency and satisfiability coincide.

Proposition 6.1.1. *If the proof system is sound and complete then consistency and satisfiability coincide.*

Proof. If T is not consistent, then for some φ, $T \vdash \varphi$ and $T \vdash \neg\varphi$. Assume towards contradiction that T is satisfiable and $\mathfrak{A}$ is a model of T, then by soundness of $\vdash$, $\mathfrak{A} \models \varphi$ and $\mathfrak{A} \models \neg\varphi$. However, no structure satisfies a formula and its negation. Contradiction.

Vice versa, if T is not satisfiable, then it holds in a trivial way that every formula is true in every model of T, hence, for arbitrary formula φ, $T \models \varphi$ and $T \models \neg\varphi$, and by completeness, $T \vdash \varphi$ and $T \vdash \neg\varphi$. We conclude that T is inconsistent. ■

Proof systems of FO In the past century, many Hilbertian proof systems for FO have been developed, consisting of different combinations of logical axiom schemata and inference rules. Also other types of proof systems have been defined such as tableaux proof method (the KE-proof method in the 1st bachelor course "Logica voor Informatici" is one), sequent calculus, natural deduction proof systems.

Standard proof systems were shown to induce the same derivability relation $\vdash$. If one is sound and complete, all of them are sound and complete. Proof of soundness of a proof theory is usually very easy. Proof of completeness is difficult. In 1930, Gödel proved this in his PhD.

Theorem 6.1.1 (Gödels Completeness theorem (1930))**.** *For each theory T and for each sentence α in FO, if $T \models \alpha$ then $T \vdash \alpha$.*

The theorem also holds if the theory T is infinite.

The completeness property can be viewed as a sign showing the strength of proof systems of FO. However, it is also a sign showing the weak expressivity of FO. Indeed, more expressive logics do not have complete proof theories.

A simple consequence of the soundness and completeness theorems is the *compactness theorem*. The compactness theorem has surprising consequences for proving inexpressivity theorems for FO as we shall see.

The compactness theorem of FO

Theorem 6.1.2 (Compactness Theorem of FO)**.** *An infinite FO theory Ψ is satisfiable iff each finite subset is satisfiable.*

Equivalently, Ψ is unsatisfiable iff at least one finite subset is unsatisfiable. It is the latter statement we prove.

Proof. ($\Leftarrow$) If Ψ has an unsatisfiable subset Ω, then Ψ is unsatisfiable, since every model of Ψ satisfies all its formulas and all its subsets.

($\Rightarrow$) If Ψ is unsatisfiable then, by Gödels completeness theorem, Ψ is inconsistent. Hence, there exists a formula φ and a proof $Pr1$ of φ from Ψ and a proof $Pr2$ of $\neg\varphi$ from Ψ.
Let Ω be $(Pr1 \cup Pr2) \cap \Psi$, the set of formulas of Ψ that are introduced in these proofs. Proofs are finite sequences; hence Ω is a finite subset of Ψ. It is obvious that $Pr1$ and $Pr2$ are also proofs from Ω, proving respectively φ and $\neg\varphi$ from Ω. Hence, Ω is inconsistent. By soundness of $\vdash$, Ω is unsatisfiable.

■

Intermezzo: Automated theorem proving Importantly, checking proofs of a proof system is mechanizable. That is, there exist algorithms that take as input any sequence of formulas and decide whether it is a proof of the given proof system. Better, there exist algorithms that search for a proof of validity of α or for a proof of α from T and that, in principle[1], will find it if one exists. This is the basis for the domain of Automated Theorem Proving.

Automated Theorem Proving (ATP) for FO is the scientific field involved in developing automated theorem provers. Building good theorem provers is very challenging. Some well-known theorem provers are Otter, SPASS, Vampire, ACL2, Waldmeister.

Theorem provers are applied for automated verification in mission critical. E.g., an early success was the use of ACL2 to prove the correctness of the floating point division operation of the AMD K5 microprocessor in the wake of the Pentium FDIV disaster. This was an event in which Pentium released a buggy floating point processor, and had to recall all of them, which costed the company around 600 million dollars.

`http://www.cs.utexas.edu/users/moore/acl2/current/manual/index.html?topic=ACL2___INTERESTING-APPLICATIONS`

The unsolved Robbins problem stated in 1933 on Boolean algebra was the first open mathematical problem that was solved by a computer, in 1996 by the theorem prover EQP in 8 days. This was news for the NY Times.

However, whereas humans lost the battle to computers in, e.g., chess and recently go, mathematicians still beat computers in solving mathematical problems, by far. In general, for creative and elegant solutions to hard mathematical and other problems, computers have not been able to beat the human mind.

Also semi-automatic theorem provers are in development; they need interaction with humans. E.g. humans guide the search for a proof of a proposition by "breaking up" the proposition in easier and useful lemmas. Well-known systems are Coq and Isabelle. They support richer logics than FO and are used more and more in program verification. They support proofs by induction which is important in verification of programs and other dynamic systems.

Intermezzo: the historical view on FO Historically, FO grew out of attempts to formalize mathematical proof. Therefore, in the view of many, FO is entangled with deductive inference. This is reflected in the fact that FO is sometimes called:

"Deductive Logic"

[1] Under the assumption of unbounded time and memory.

Deductive Logic was seen as a trinity:

- a *formal syntax*;
- a *proof theory*: formal definition of a proof system and the notion of proof.
- a *model theory*: formal definition of semantical entailment: $T \models \alpha$ means that α is true in all models of T.

Deductive inference is an important form of inference but we realize now that many if not most practical problems can be solved using simpler and cheaper forms of inference.

6.2 Some expressivity limitations of FO

The expressivity analysis of logic $\mathcal{L}$ is the scientific study of what propositions can be expressed in $\mathcal{L}$. We explain how it works with an example.

Example 6.2.1. Consider the informal proposition

$$\Phi = \text{“}\textbf{The vertex A has no outgoing edges in graph G}\text{”}.$$

The proposition is informal because it is in natural language. It talks about two mathematical concepts: a vertex A and a graph G. No value for these concepts is specified here. A state of affairs for this proposition corresponds to a tuple ⟨G,A⟩ of a graph G and a vertex A. Although Φ is informal, it is nevertheless mathematically precise in the sense that in any state of affairs ⟨G,A⟩, this sentence is either true or false. Thus, Φ determines the class $\mathcal{C}_\Phi$ of all states of affairs ⟨G,A⟩ where Φ is true. This $\mathcal{C}_\Phi$ formalizes the information content of Φ, in the same way as the class of solutions of Newtons theory of gravitation formalizes the information content of this scientific theory (confer Chapter 1).

Now the question arises : can we express Φ in a logic $\mathcal{L}$ (with a model theoretic semantics)? The question amounts to the following: can we write a logic theory T in some vocabulary Σ such that the class $Mod(T)$ of models of T is ($\pm$) the same as $\mathcal{C}_\Phi$?

First, we need an appropriate choice of Σ so that Σ-structures correspond to states of affairs for Φ. In the example, a good choice is $\Sigma = \{A/0 :, G/2\}$ together with the informal interpretation $\mathcal{I}$ linking symbols of Σ to the concepts in Φ: constant A is informally interpreted as the vertex A and binary predicate G is informally interpreted as the graph G. [2] Let $\mathcal{C}_\Sigma$ be the class of structures interpreting Σ. Every structure $\mathfrak{A} \in \mathcal{C}_\Sigma$ specifies a state of affairs $\langle G^{\mathfrak{A}}, A^{\mathfrak{A}} \rangle$ for Φ. Vice versa, each state of affairs ⟨G,A⟩ corresponds to a structure $\mathfrak{A}$ such that $G^{\mathfrak{A}}$ =G and $A^{\mathfrak{A}}$ =A. So, there is a match between the structures of Σ and states of affairs for Φ. From now on, we view $\mathcal{C}_\Phi$ as the corresponding class of Σ-structures.

With Σ and $\mathcal{I}$ in the back of our head, the question now is: can we find a theory T such that $Mod(T)$, the class of models of T, corresponds to $\mathcal{C}_\Phi$? I.e., $\mathcal{C}_\Phi = \{\mathfrak{A} \in \mathcal{C}_\Sigma \mid \mathfrak{A} \models \phi\}$.

In this particular case, the answer is positive. Φ is expressed by:

$$\neg\exists x\ G(A, x)$$

[2] The name of a symbol of Σ does not matter; what matters is that its type matches the type of its informal interpetation and that we remember its informal interpretation. Of course, for convenience, it is best to use a symbol σ that syntactically explains which concept $\sigma^{\mathcal{I}}$ it represents.

We show that it expresses the class $\mathcal{C}_\Phi$. Indeed, $\mathfrak{A} \models \neg\exists x\ G(A, x)$ iff there exists no vertex $d \in D_\mathfrak{A}$ such that there is an edge in $G^\mathfrak{A}$ from $A^\mathfrak{A}$ to d iff A has zero outgoing edges in $G^\mathfrak{A}$ iff $\mathfrak{A} \in \mathcal{C}_\Phi$.

Summarized, the method works as follows.

- We want to analyze a precise informal proposition Φ about a collection of named but unspecified mathematical objects (atomic objects, sets, relations, functions). Φ determines the class $\mathcal{C}_\Phi$ of states of affairs (tuples of values for the mathematical concepts in Φ) in which Φ is true. This class $\mathcal{C}_\Phi$ is the formalization of the information content of Φ.
- The first step is to choose a formal vocabulary Σ and an informal interpretation $\mathcal{I}$ linking symbols of Σ with concepts in Φ so that Σ-structures correspond to states of affairs. Let $\mathcal{C}_\Sigma$ be the class of structures interpreting Σ, then $\mathcal{C}_\Phi$ corresponds to a subclass $\mathcal{C}$ of $\mathcal{C}_\Sigma$.
- Prove that there exists or does not exist a formula or theory ϕ in $\mathcal{L}$ such that $\mathfrak{A} \models \phi$ iff $\mathfrak{A} \in \mathcal{C}$. Proving that Φ exists is usually done by providing a ϕ that expresses Φ. Proving that such a ϕ does not exist requires more complex methods. Such a proof shows that Φ cannot be expressed in the logic.

Example 6.2.2. Consider the informal proposition:

$$\Phi = \text{“}\textbf{The graph G is reflexive}\text{”}$$

- This proposition talks about a graph G. As a formal vocabulary, we take $\Sigma = \{G/2\}$.
- The corresponding class $\mathcal{C}$ consists of structures $\mathfrak{A}$ that interpret Σ and in which $G^\mathfrak{A}$ is a reflexive relation; i.e., $\{(d, d) \mid d \in dom_\mathfrak{A}\} \subseteq G^\mathfrak{A}$.
- Φ is expressed by
$$\forall x\ G(x, x)$$

  Proof. Every structure $\mathfrak{A}$ interpreting Σ, satisfies $\forall x\ G(x, x)$ iff for each $d \in D_\mathfrak{A}$, $G^\mathfrak{A}$ contains (d, d) iff $G^\mathfrak{A}$ is reflexive iff $\mathfrak{A} \in \mathcal{C}$. ■

(In)expressivity results An expressivity result of an informal proposition Φ is shown by providing a concrete formula or theory that expresses the proposition. E.g., reflexivity.

An inexpressivity result is more difficult to prove, since we need to prove for the infinite set of formulas or theories that none expresses the informal proposition Φ.

The compactness theorem was the first technique that was used to prove inexpressivity results. We will use it to prove inexpressivity in FO of three propositions that have appeared earlier in this course.

Basic terminology and notations For logic formula φ over Σ, $Mod(\varphi)$ denotes the class of models of φ. Likewise, for theory T, $Mod(T)$ denotes the class of models of T.

Recall that the class of structures that interpret at least the symbols of Σ is denoted $\mathcal{C}_\Sigma$.

Let $\mathcal{C}, \mathcal{C}'$ be classes of structures. We say that $\mathcal{C}$ is inconsistent with $\mathcal{C}'$ if $\mathcal{C} \cap \mathcal{C}' = \emptyset$. We say that $\mathcal{C}$ is inconsistent with a theory or formula T if $\mathcal{C}$ is inconsistent with $Mod(T)$.

Simple property: $\mathfrak{A} \models T \cup T'$ iff $\mathfrak{A} \models T$ and $\mathfrak{A} \models T'$ iff $\mathfrak{A} \in Mod(T) \cap Mod(T')$.

The classes $\mathcal{C}$ of structures that we consider in this section satisfy two natural conditions:

- $\mathcal{C}$ is closed under isomorphism: if $\mathfrak{A}$ is isomorphic to $\mathfrak{A}'$ then $\mathfrak{A} \in \mathcal{C}$ iff $\mathfrak{A}' \in \mathcal{C}$.
- $\mathcal{C}$ is closed under extension: if $\mathfrak{A} \in \mathcal{C}$ and $\mathfrak{A}'$ extends $\mathfrak{A}$ with an interpretation for additional symbols, then $\mathfrak{A}' \in \mathcal{C}$.

Notice that for any formula or theory T, the class $Mod(T)$ satisfies these conditions. Hence, FO can only express classes of structures that satisfy these conditions.

Expressing classes of structures

Definition 6.2.1.
- A formula φ of logic $\mathcal{L}$ expresses a class $\mathcal{C}$ of structures if $Mod(\varphi) = \mathcal{C}$.
- A theory T of logic $\mathcal{L}$ expresses $\mathcal{C}$ if $Mod(T) = \mathcal{C}$.
- A class $\mathcal{C}$ is finitely expressible in $\mathcal{L}$ if it is expressed by a formula φ of $\mathcal{L}$.
- A class $\mathcal{C}$ is expressible in $\mathcal{L}$ if it is expressed by a (possibly infinite) theory T of $\mathcal{L}$.

An example: a proposition expressed by an infinite theory

"The universe is infinite".

This proposition only talks about the domain of a structure, which exists in every structure. Hence, we take $\Sigma = \emptyset$, which means that the only predicate symbol that can be used in formulas to express this is the logical equation symbol.

Its class, denoted $\mathcal{C}_{InfU}$, is the class of structures with infinite domain.

It is expressed by the following theory $T_{InfU} =$

$$\left\{ \begin{array}{l} \exists x_1 \exists x_2 (\neg(x_1 = x_2)), \\ \exists x_1 \exists x_2 \exists x_3 (\neg(x_1 = x_2) \wedge \neg(x_1 = x_3) \wedge \neg(x_2 = x_3)), \\ \ldots \\ \exists x_1 \ldots \exists x_n (\neg(x_1 = x_2) \wedge \cdots \wedge \neg(x_1 = x_n) \wedge \neg(x_2 = x_3) \wedge \cdots \wedge \neg(x_{n-1} = x_n)), \\ \ldots \end{array} \right\}$$

The latter expression contains a non-equality $\neg(x_i = x_j)$ for every $i < j$ in $[1, n]$. The formula expresses that (at least) n different objects can be found in the domain.

Proof. This theory consists of axioms expressing: there are least 2 domain elements, at least 3, at least 4, ... Any structure with an infinite domain satisfies each of these propositions. Vice versa, each structure that satisfies all these propositions has an infinite domain. ■

Formulas are finite, theories may be infinite. Therefore, we should expect that we can express more with theories than with formulas. In fact, we will prove later that $\mathcal{C}_{InfU}$ cannot be finitely expressed (see Corollary 6.2.2).

Theorem 6.2.1 (Inexpressivity theorem)**.** *Let Σ be a vocabulary, let $\mathcal{C}$ be a class of structures interpreting Σ ($\mathcal{C} \subseteq \mathcal{C}_\Sigma$), let T' be an infinite FO theory over Σ such that:*

1. *T' is inconsistent with $\mathcal{C}$;*
2. *each finite subset of T' is consistent with $\mathcal{C}$.*

Then $\mathcal{C}$ is not expressible in FO.
(Moreover, T' is not finitely expressible: see Corollary 6.2.2).

In mathematical terms, the conditions to be satisfied by T' are:

1. $\mathcal{C} \cap Mod(T') = \emptyset$;
2. $\mathcal{C} \cap Mod(\Omega) \neq \emptyset$, for each finite subset $\Omega \subseteq T'$.

Some explanation about the theorem: think of $\mathcal{C}$ as the class of Σ-structures satisfying the informal but precise proposition Φ that we introduced before.

Also the *infinite* FO theory T' expresses an informal proposition (through its informal interpretation), say Ψ.

So, we have two informal propositions, Φ and Ψ, the second one being expressed by the infinite FO theory T'.

The first condition $\mathcal{C} \cap Mod(T') = \emptyset$ means that the conjunction of Φ and Ψ is inconsistent; there are no structures satisfying both.

The second condition, $\mathcal{C} \cap Mod(\Omega) \neq \emptyset$ for every finite subtheory Ω of T', means that Φ is consistent with *every* *finite* subtheory Ω of T'.

The theorem then ensures that under these conditions, Φ is not expressable in FO.

Proof. Assume towards contradiction that $\mathcal{C}$ is expressed by FO theory T; i.e., $Mod(T) = \mathcal{C}$.

It follows that $Mod(T) \cap Mod(T') = \emptyset$, hence $T \cup T'$ is unsatisfiable.

By the compactness theorem, $T \cup T'$ has an unsatisfiable finite subset $\Omega \subseteq T \cup T'$.

Consider $\Omega' = \Omega \cap T'$, a finite subtheory of T'. Elements of $\Omega \setminus \Omega'$ belong to T. Hence, it holds that $\Omega' \subseteq \Omega \subseteq \Omega' \cup T$.

On the one hand, Ω' is a finite subtheory of T', hence $\Omega' \cup T$ is satisfiable (since $Mod(\Omega' \cup T) = Mod(\Omega') \cap \mathcal{C} \neq \emptyset$).

On the other hand, $\Omega' \cup T$ is a superset of Ω. Any superset of an unsatisfiable FO theory is unsatisfiable as well, hence $\Omega' \cup T$ is unsatisfiable. Contradiction. ■

To apply this theorem to prove that some informal but precise proposition Φ about concepts expressed in vocabulary Σ cannot be expressed in FO, find another property Ψ such that

- Φ and Ψ contradict each other: there are no structures satisfying both. E.g., Ψ might be the negation of Φ.
- Ψ can be expressed by an infinite FO theory T'.
- But Φ is consistent with every finite subtheory Ω of T'.

In that case, the theorem ensures that Φ cannot be expressed in FO. In addition, we will show later that T' is not finitely expressible. Thus, there is a gap between infinite and finite expressibility in FO. But first, we consider three applications of the inexpressvity theorem.

Inexpressivity result 1: Finiteness of the universe cannot be expressed

Theorem 6.2.2. *"**The universe is finite**" is not expressible in FO.*

Proof. This informal proposition characterises the class of structures with a finite universe, which we denote $\mathcal{C}_{FinU}$. Consider the class $\mathcal{C}_{InfU}$ of structures with infinite universe. It is expressed by T_{InfU}, as shown on p. 193. Let us verify that the conditions of the theorem are satisfied.

- $\mathcal{C}_{FinU} \cap \mathcal{C}_{InfU} = \emptyset$: obviously.
- Each finite subset Ω of T_{InfU} is consistent with $\mathcal{C}_{FinU}$. Indeed, there is a largest number n such that Ω contains the axiom from T_{InfU} that expresses that the domain contains at least n elements. Any finite structure with domain size n or more satisfies Ω.

Application of the inexpressivity theorem yields that $\mathcal{C}_{FinU}$ is not expressible in FO. ■

The proposition "The universe is finite" can be expressed in SO. It exploits the fact that a set is infinite if and only if there exists an injective mapping from the set to a strict subset of it. The following SO formula expresses that there is an injective function F that maps the universe of the structure to a strict subset of it:

$$\exists F[(\forall x \forall y(\neg(x = y) \Rightarrow \neg(F(x) = F(y)))) \wedge (\exists y \neg \exists x F(x) = y)]$$

The negation of this SO formula is a way to express that the universe is finite.

Exercise 6.2.1. *Express the stronger property that the universe contains at most N elements in FO.*

Remark 6.2.1. Applications of the inexpressivity theorem for proving inexpressivity of a proposition are frequent exam questions. A valid proof of inexpressivity of some Φ comprises the definition of the corresponding infinite theory T' and a proof that the conditions on T' and $\mathcal{C}$ are satisfied. If this is missing, there is no valid proof. E.g., to prove inexpressivity of "the universe is finite", specify T_{InfU} and show that the two conditions of the inexpressivity theorem hold.

Inexpressivity result 2: Reachability cannot be expressed in FO Consider first the proposition "**There is no path from A to B in graph G**". We show that it can be infinitely expressed in FO.

- A suitable vocabulary is $\Sigma = \{A, B, G/2\}$.
- The proposition is then expressed by the theory $T_{NoPath} =$

$$\left\{ \begin{array}{l} \neg G(A,B), \\ \neg\exists x_1(G(A,x_1) \wedge G(x_1,B)), \\ \ldots \\ \neg\exists x_1 \ldots \exists x_n(G(A,x_1) \wedge \cdots \wedge G(x_n,B)), \\ \ldots \end{array} \right\}$$

Proof. The propositions express: there is no path of length 1, of length 2, of length 3, ... A structure in which there is no path from A to B satisfies each of these propositions, and vice versa, a structure that satisfies each of these propositions cannot have a path from A to B. ■

We will use T_{NoPath} in the following theorem, showing that the negation of the above informal proposition cannot be expressed in FO.

Theorem 6.2.3. *"***There is a path from A to B in graph G***" is not expressible in FO.*

Proof. The proposition characterises the class of structures $\mathfrak{A}$ interpreting G by a graph with a path from $A^{\mathfrak{A}}$ to $B^{\mathfrak{A}}$. We denote this class as $\mathcal{C}_{Path}$. This class is inconsistent with T_{NoPath} but consistent with each finite subset Ω of T_{NoPath}. Indeed, let n be the largest path length forbidden by Ω. Every structure with a shortest path from A to B that is strictly longer than n satisfies Ω and belongs to $\mathcal{C}_{Path}$. The inexpressivity theorem applies. ■

Reachability, transitive closure are important concepts in many applications, but they cannot be expressed in FO.

Exercise 6.2.2. *Express the proposition "There is a path from A to B in graph G" in SO. Hint: express that (A,B) is contained in every transitive relation that extends G. Explain why this is correct.*

Strictly speaking, the property cannot be expressed in FO(.). However, it can be expressed if we may introduce an auxiliary defined symbol. The FO(.) theory is:

$$\left\{ \begin{array}{l} \forall x \forall y(Path(x,y) \leftarrow G(x,y)) \\ \forall x \forall y(Path(x,y) \leftarrow \exists z(Path(x,z) \wedge Path(z,y))) \end{array} \right\}$$
$$Path(A,B)$$

This theory expresses "*Path is the transitive closure of G and* (A,B) *is in Path*". This entails that "*there is a path from A to B in G*", but it is stronger in the sense that it also specifies that $Path$ is the transitive closure of G. However, there is a one-to-one correspondence between the models of this theory and the class of structures (not interpreting $Path$) in which there is a path from A to B. That is good enough in practice.

Inexpressivity result 3: the Domain Closure Axiom is inexpressible Let τ be a finite set of constant symbols and function symbols. Let S_τ be the set of terms over τ.

The **domain closure axiom of** τ is the informal proposition that each element of the universe is represented by a term of S_τ. This proposition is denoted as $\mathbf{DCA}(\tau)$.

Recall that the universe (also called domain) of a structure $\mathfrak{A}$ is denoted as $D_\mathfrak{A}$. A structure $\mathfrak{A}$ satisfies $\mathbf{DCA}(\tau)$ iff

$$D_\mathfrak{A} = \{t^\mathfrak{A} \mid t \in S_\tau\}$$

Hence, the class $\mathcal{C}_{DCA(\tau)}$ characterized by $\mathbf{DCA}(\tau)$:

$$\mathcal{C}_{DCA(\tau)} = \{\mathfrak{A} \mid D_\mathfrak{A} = \{t^\mathfrak{A} \mid t \in S_\tau\}\}$$

Motivation In some programming languages like Prolog or Haskell, the domain of objects of a program is fixed to be the class of terms of the set τ of constructor symbols used in the program.

As we have seen, e.g., in the context of databases, FO is more liberal and accepts structures in which the universe is not the set of terms. The effect to this is that uncertainty on the domain can be represented. In some applications, the user has uncertainty on the domain, and then it is useful that this uncertainty can be expressed in the FO theory. But in other applications, the user has complete knowledge of the domain and knows the domain is the set of terms over some vocabulary τ. In that case, it would be useful to be able to express this knowledge in the logic. In FO, we would need to express this by a combination of UNA (terms denote different objects) and DCA (there are no domain elements not denoted by a term). But can we do this?

In Chapter 2, we saw how to express UNA for arbitrary τ in FO and DCA if τ is a finite set of constant symbols. Now we prove that $\mathbf{DCA}(\tau)$ cannot be expressed in FO if τ contains function symbols.

If τ contains no constant symbols, then $S_\tau = \emptyset$: no terms can be built from τ and the domain closure is unsatisfiable (the universe of a structure of FO is not allowed to be empty).

If τ is a set $\{C_1, \ldots, C_n\}$ of constants then $S_\tau = \tau$. We already saw how to express $\mathcal{C}_{DCA(\tau)}$ in FO.

If $\tau = \{0, S/1 :\}$, then $S_\tau = \{0, S(0), S(S(0)), \ldots\}$. The domain closure axiom corresponds to the *induction axiom* which can be expressed in second order logic by Peano's induction axiom. The question now is: can we express $\mathcal{C}_{DCA(\tau)}$ in FO?

Theorem 6.2.4. *If τ contains at least one constant and one function symbol then* $\mathbf{DCA}(\tau)$ *is not expressible in FO.*

Proof. Let a be a new constant **not** in τ. Hence $a \notin S_\tau$.

Consider the infinite theory $T_a = \{\neg(a = t) \mid t \in S_\tau\}$. This theory states that a is an object that is different from each term in S_τ. For any model $\mathfrak{A}$ of T_a, it holds that $a^\mathfrak{A} \in D_\mathfrak{A} \setminus \{t^\mathfrak{A} \mid t \in S_\tau\}$; such a structure does not satisfy $\mathbf{DCA}(\tau)$. Hence, $\mathcal{C}_{DCA(\tau)}$ is inconsistent with T_a.

However, $\mathcal{C}_{DCA(\tau)}$ is consistent with each finite subset, and even with each strict subset of T_a. Indeed, take a Herbrand interpretation $\mathfrak{A}$ of τ. It holds that for any pair of terms t, s over τ that $t^{\mathfrak{A}} \neq s^{\mathfrak{A}}$. Let Ω be a strict subset of T_a. There is at least one term $t \in S_\tau$ such that $\neg(a = t) \notin \Omega$. Take the structure $\mathfrak{A}[a : t^{\mathfrak{A}}]$ that expands $\mathfrak{A}$ by interpreting a as $t^{\mathfrak{A}}$. For each axiom $\neg(a = t') \in \Omega$, it holds that t' is a different term than t and hence, $a^{\mathfrak{A}[a:t^{\mathfrak{A}}]} = t^{\mathfrak{A}} \neq t'^{\mathfrak{A}}$. Hence, $\mathfrak{A}[a : t^{\mathfrak{A}}]$ satisfies Ω.

Applying the inexpressivity theorem now yields the theorem.

■

Exercise 6.2.3. *The proof is not correct if τ does not contain at least one function symbol of arity > 0. Why not?*

For $\tau = \{0, S/1\}$, the domain closure axiom says something like:

"**the universe consists of 0, 1, 2,. . . , n, n+1,. . . , and nothing more.**"

This is not a difficult proposition to understand for us. Even children understand it. As a PhD student, I was greatly surprised (and disappointed) in FO to discover that such a simple proposition could not be expressed in FO.

Finite versus infinite expressivity in FO The compactness theorem is also useful to show that not all infinitely expressible propositions in FO are finitely expressible in FO. In particular, we can show that the infinite theory T' used in the inexpressivity theorem is not finitely expressible.

Corollary 6.2.1. *Let T' be an infinite FO theory over Σ such that each finite subtheory Ω of T' has models that do not satisfy T'. Then T' is not finitely expressible in FO and the negation of T' is not expressible in FO.*

Proof. Take the class $\mathcal{C}_{\neg T'} = \mathcal{C}_\Sigma \setminus Mod(T')$ consisting of structures that do not satisfy T'. We show that $\mathcal{C}_{\neg T'}$ and T' satisfy the conditions of the inexpressivity theorem.

1. $\mathcal{C}_{\neg T'} \cap Mod(T') = \emptyset$; obvious, since no structure can both satisfy a theory and not satisfy it.
2. Each finite subset $\Omega \subseteq T'$ has models not satisfying T; they belong to $\mathcal{C}_{\neg T'}$. It follows that $\mathcal{C}_{\neg T'} \cap Mod(\Omega) \neq \emptyset$.

It follows from the expressivity theorem that $\mathcal{C}_{\neg T'}$ is not expressible in FO. Thus, the negation of T' is not expressible in FO.

Suppose that T' would be finitely expressible in FO, then it would be expressed by some FO formula ϕ. But then the negation of T' would be expressed by $\neg\phi$. Contradiction. So, T' is not finitely expressible.

■

Corollary 6.2.2. *Under the same conditions for $\mathcal{C}$ and T' as in the inexpressivity theorem, the negation of T' is not expressible, and T' is not finitely expressible.*

Proof. Each finite subset $\Omega \subseteq T'$ has some models in $\mathcal{C}$ and they are not models of T'. Now we apply Corollary 6.2.1 to obtain that T' is not finitely expressible and the negation of T' is not expressible in FO. ■

Here we have a technique to achieve two things:

- To prove that some informal proposition is not expressible in FO.
- To prove that some infinitely expressible proposition is not finitely expressible in FO.

This theorem suggests that the negation of an infinitely expressible informal proposition is a good candidate for not being expressible in FO. Indeed, such an informal statement satisfies already the first condition of the theorem.

E.g., "the universe is infinite" is infinitely expressible (it is expressed by T_{infU} as proven). Its negation is ... "the universe is finite".

Non-standard models of FO theories of the natural numbers Recall Peano's vocabulary $\Sigma_P = \{0, S/1 :, +/2 :, \times/2 :\}$. The standard structure of natural numbers over this vocabulary is denoted by $\mathbb{N}$.

Theorem 6.2.5. *Every FO theory T over Σ_P that is satisfied in $\mathbb{N}$ has Σ_P-models that are not isomorphic with $\mathbb{N}$.*

Stated differently, the class of Σ_P-structures isomorphic to $\mathbb{N}$ is not expressible in FO.

Proof. It suffices to show that every FO theory satisfied in $\mathbb{N}$ has non-countable models. Indeed, non-countable models are not isomorphic to the countable structure $\mathbb{N}$.

Let A be a non-countable set of new constants, and D_A the theory $\{\neg(c = d) | c, d \in A\}$ expressing that all these constants are different. Any model of D_A is clearly non-countable.

Take any finite subtheory Ω of $T \cup D_A$. It contains only a finite number of disequalities of D_A. It is easy to see that we can extend $\mathbb{N}$ with values for constants in A so that the disequalities in $\Omega \cap D_A$ are satified. The resulting structure is clearly a model of Ω, hence Ω is satisfiable. The compactness theorem now tells us that $T \cup D_A$ is satisfiable, hence it has non-countable models of T.

■

Corollary 6.2.3. *The FO theory PA (Peano arithmetic with the induction schema) has Σ_P-models that are not isomorphic to $\mathbb{N}$.*

Such non-isomorphic models are called *non-standard models*.

In contrast, any models of the SO theory that we have called the Peano axioms, is isomorphic to $\mathbb{N}$. This theorem was given in Chapter 2.

Even the infinite FO theory consisting of *all* formulas that are true in $\mathbb{N}$ has models that are not isomorphic with $\mathbb{N}$.

Corollary 6.2.4. *The theory $\mathtt{Th}(\mathbb{N}) = \{\alpha \mid \mathbb{N} \models \alpha\}$ has Σ_P-models that are not isomorphic to $\mathbb{N}$.*

Both corollaries follow immediately from Theorem 6.2.5.

The conclusion is that in FO, one cannot express all information in the structure $\mathbb{N}$. The culprit is the domain closure axiom.

6.2.1 Conclusion

We saw three implications of the compactness theorem:

- We cannot express *finiteness* of the universe.
- We cannot express *reachability*.
- We cannot express *domain closure*.

These propositions are important in applications. This motivates to extend FO with additional language constructs.

In the same effort, we also obtained proofs that the following propositions are infinitely but not finitely expressible in FO:

- the universe is infinite,
- there is no path from A to B in graph G,
- a is different from each term in S_τ.

6.2.2 Intermezzo: Additional results

- "**Binary relation Tr is the transitive closure of G**" cannot be expressed.

Proof. The class $\mathcal{C}$ of this informal proposition is inconsistent with the theory $\{Tr(A,B)\} \cup T_{NoPath}$ but $\mathcal{C}$ is consistent with each finite subset of this theory. (For the definition of T_{NoPath} see page 196). ■

- "**G is a connected graph**" cannot be expressed.

Proof. The corresponding class $\mathcal{C}$ of structures is inconsistent with T_{NoPath} (which states that A and B are not connected). However, $\mathcal{C}$ is consistent with each finite subset of T_{NoPath}. ■

Exercise 6.2.4. *Express the proposition "***The graph G is acyclic***" in FO and show that the proposition "***The graph G is cyclic***" is not expressible in FO. Express it in SO or FO(.).*

Exercise 6.2.5. *Prove that the proposition "***The graph G is connected***" is not expressible in FO. Express it in SO or FO(.).*

The compactness theorem is historically the first technique to prove that a property cannot be expressed in FO. More advanced techniques exist. E.g., Ehrenfeucht-Fraïssé Games.

6.3 Undecidability and Gödels incompleteness theorem

Recall from Definition 1.3.1 the notions of (semi-)decidability of computational problems $\mathcal{M}$.

First question The deductive inference problem is the decision problem $\mathcal{M}_{valid}$ that takes as input a finite theory T and formula φ and returns **t** if $T \models \varphi$, and **f** otherwise. The validity checking problem is the decision problem that takes as input a formula φ and returns **t** if $\models \varphi$, and **f** otherwise.

The validity problem with input φ can be reduced to the deductive inference problem by taking $T = \emptyset$. Vice versa, the deduction inference problem for input T, φ can be reduced to the validity problem for input $(\wedge_{\psi \in T} \psi) \Rightarrow \varphi$. Thus, these two problems are two sides of the same coin.

Now the question rises: are these problems decidable?

Theorem 6.3.1 (Undecidability of FO)**.** *The validity inference problem for FO is undecidable.*

This theorem was proven by Church in 1936 and independently by Turing in 1937.

Proof. We only give the idea of the proof. Take an undecidable decision problem $\mathcal{M}$ and design a translation from any input i for this problem to a formula φ_i such that $\mathcal{M}(i) = \mathbf{t}$ iff φ_i is valid. Thus, $\mathcal{M}$ can be reduced to $\mathcal{M}_{valid}$, the validity checking inference problem. It follows that $\mathcal{M}_{valid}$ is undecidable. [3] ■

[3] Many proofs exist making different choices of $\mathcal{M}$, e.g., the Post Correspondence problem or the Halting problem, or the acceptance problem $\{(M,w) | M \text{ is a Turing machine that accepts string } w\}$. For an example, see `http://kilby.stanford.edu/~rvg/154/handouts/fol.html`.

Theorem 6.3.2 (Semi-decidability of FO)**.** *The validity problem for first order logic is semi-decidable.*

I.e., there is an algorithm that for input φ answers **t** if $\models \varphi$ and otherwise returns **f** or does not terminate.

Proof. The basic idea is simple. A sketch is as follows. According to Gödels completeness theorem, there exists a sound and complete proof system for FO. Proofs in this proof system can be encoded as finite strings of some vocabulary (e.g., $\Sigma = \{0, 1\}$ will suffice) such that there is a terminating algorithm A to decide for any input string in Σ^* whether it is a proof of that proof system or not.

Now consider the following algorithm: generate exhaustively all strings over Σ; for each generated string w, run A to check if w is a proof and if so, check if the last proposition in w is equal to the input formula. If so, stop and return **t**. Otherwise, go on.

If the input formula is valid, a proof exists and its encoding will eventually be generated. So the algorithm will terminate with **t**. Otherwise, the algorithm will not terminate. ■

We summarize these properties by stating that deductive inference in FO is undecidable but semi-decidable.

Second question Can we build an FO theory from which every true FO sentence in $\mathbb{N}$ can be proven?

Well, sure! Take $\mathit{theory}(\mathbb{N}) = \{\varphi \mid \mathbb{N} \models \varphi\}$. It contains every such sentence and hence entails it.

However, $\mathit{theory}(\mathbb{N})$ is clearly infinite. Can we find a finite theory? Or, more generally, can we find a *finitely representable* theory? To answer that question, we first need to formally define when a theory T is finitely representable? A most general answer is when the axioms of T can be *generated by an algorithm*. That is, if T is *recursively enumerable*. In this case, the finite representation of T is the algorithm that generates it. E.g., Peano arithmetic, the FO theory PA with the induction schema is infinite, but it is finitely representable by six axioms and one axiom schema. Intuitively it is clear that an algorithm exists to generate all instances of the axiom schema: generate all formulas and plug them in in the schema. Hence, PA is recursively enumerable.

The refined question then is: can we build a recursively enumerable FO theory that is satisfied in $\mathbb{N}$ and from which every true FO sentence in $\mathbb{N}$ can be proven? That question was solved by Gödel.

Gödels incompleteness theorem (1931)

Theorem 6.3.3 (Gödels incompleteness theorem (1931))**.** *For any recursively enumerable FO theory T that is satisfied in $\mathbb{N}$, there exists an FO sentence α such that α is true in $\mathbb{N}$ but $T \not\models \alpha$.*

Stated alternatively, a satisfiable, recursively enumerable theory of FO from which every FO sentence true in $\mathbb{N}$ can be proven, does not exist.

The condition that T is recursively enumerable ensures the effective finite representability of the theory. The FO theory of Peano arithmetic (with induction schema) is recursively enumerable and is satisfied in the natural numbers. Hence, the theorem entails that there exist sentences true in $\mathbb{N}$ but not provable from this theory.

Recall: in the inexpressivity section, we proved that every FO theory satisfied by $\mathbb{N}$ has unintended models (not isomorphic to $\mathbb{N}$). Nevertheless, there exist FO theories that entail all FO sentences that are true in $\mathbb{N}$! An example is the theory $\mathit{theory}(\mathbb{N}) = \{\varphi \mid \mathbb{N} \models \varphi\}$. It contains every sentence true in $\mathbb{N}$ and hence entails it.

What does Gödels incompleteness theorem say about the latter sort of theories? Well, since such a theory contradicts the conclusion of the theorem, it must violate the premise of the theorem. Such a theory is not recursively enumerable.

Corollary 6.3.1. *If T is satisfied by $\mathbb{N}$ and entails all formulas φ true in $\mathbb{N}$, then T is not recursively enumerable. In particular, $\mathit{theory}(\mathbb{N})$ is not recursively enumerable.*

In other words, $\mathit{theory}(\mathbb{N})$ is not semi-decidable: there is no algorithm that for input φ, answers whether $\varphi \in \mathit{theory}(\mathbb{N})$ and that is guaranteed to terminate if the answer is positive. Notice that $\varphi \in \mathit{theory}(\mathbb{N})$ if and only if $\mathbb{N} \models \varphi$.

Corollary 6.3.2 (undecidability of the natural numbers)**.** *The problem of deciding truth of an FO sentence in $\mathbb{N}$ is not decidable and not even semi-decidable.*

This property is known as the *undecidability of the natural numbers.* It is a direct consequence of Gödels incompleteness theorem.

Corollary 6.3.3. *Any extension $\mathcal{L}$ of FO with a finite (or recursively enumerable) theory $T_{\mathbb{N}}$ of $\mathbb{N}$ that is categorical, does not have a sound and complete proof system.*

Recall, a theory is categorical if all its models are isomorphic.

Proof. Assume towards contradiction that $\mathcal{L}$ has a sound and complete proof system. We build a decision procedure for $\mathbb{N}$ as follows. Take the algorithm that takes as input an FO formula φ over Σ_P and runs an algorithm that generates all proofs from $T_{\mathbb{N}}$ in the proof system of $\mathcal{L}$. If φ is proven, it returns **t** and stops; if $\neg\varphi$ is proven then it returns **f**. Since either $\mathbb{N} \models \varphi$ or $\mathbb{N} \models \neg\varphi$, sooner or later one of the two will be proven. Hence, the algorithm terminates with the correct answer. We built a decision procedure for $\mathbb{N}$. Contradiction. ■

So far, we have seen two logics in which we can express the structure of the natural numbers: second order logic (SO) (confer Peano's theory with the second order induction axiom) and FO(.)

in which the induction axiom can be expressed as well. They have no sound and complete proof system.

Historical note Gödel did not present his theorem in the above form. E.g., he stated his theorem in terms of $\vdash$ (derivability) instead of logical entailment $\models$ (but both relations are identical for FO). Also, he did not use the concept of "recursive enumerable" which did not exist yet in 1931. In fact, he proved his theorem only for a specific FO theory, namely Russell and Whitehead's Principia Mathematica. However, in the introduction he had explained that his proof could be generalized for any theory that could somehow be "generated".

Largely inspired by his work, Church (1936) and Turing (1937) developed formal definitions of computability, and were able to state and prove the undecidability of validity and the natural numbers.

Intermezzo: Gödels proof His proof was brilliant (and extremely tedious). He showed how to encode formulas by numbers

$$\varphi \longrightarrow n_\varphi$$

He constructed a formula $\varphi_T[n]$ that is true for a number n iff this number is the representation of a provable sentence.

He then showed the existence of a self-referential FO sentence, called the *Gödel sentence* which intuitively means:

"*I cannot be proven*"

More precisely, he proved that there is a number m that encodes the sentence $\neg\varphi_T[m]$. (i.e. $n_{\neg\varphi_T[m]} = m$).

This sentence, like any other is either true or false in $\mathbb{N}$. If it would be false, then its negation $\varphi_T[m]$ would be true. By construction of φ_T, the formula encoded by m (i.e., $\neg\varphi_T[m]$) is provable from T and hence, by the soundness of $\vdash$, true in $\mathbb{N}$. This is a contradiction. Hence, $\neg\varphi_T[m]$ must be true.

Since this sentence is true, it cannot be proven. Hence, this is a true but unprovable sentence.

Intermezzo: impact of the theorem Gödels incompleteness theorem is perhaps the most famous theorem of mathematical logic.

In mathematics, the meaning of Gödels incompleteness theorem has been immense. Before Gödel, mathematicians hoped that they could axiomatize every useful structure (such as the natural numbers) and could prove the consistency of the theory and all its theorems. This was called

"Hilbert's program"

A crucial step in this program was the *Entscheidungsproblem*, the problem of finding an algorithm for deciding validity or satisfiability. Gödel proved that this problem was unsolvable and hence, that Hilbert's program could not be realised.

The Classical "Entscheidungsproblem" The question whether deciding validity in first-order logic is possible was an important open problem in mathematical logic.

> *Das Entscheidungsproblem ist gelöst, wenn man ein Verfahren kennt, das bei einem vorgelegten logischen Ausdruck durch endlich viele Operationen die Entscheidung über die Allgemeingültigkeit bzw. Erfüllbarkeit erlaubt. (...) Das Entscheidungsproblem muss als das Hauptproblem der mathematischen Logik betrachtet werden.*

(D. Hilbert, W. Ackermann: Grundzüge der theoretischen Logik, Vol. 1, Berlin 1928, p.73)

> *The "decision problem" is solved, if one knows an algorithm which, given a logical expression, decides by finitely many operations, if the expression is satisfiable or valid, resp. (...) The "decision problem" must be seen as the most important problem in mathematical logic.*

Gödel showed that this problem is unsolvable.

Intermezzo: impact of the theorem in AI In philosophical AI, Gödels incompleteness theorem has given rise to speculation regarding the adequacy of *logic* and the standard *Von Neumann computer architecture* for building AI-systems.

Some have argued along the following lines:

- Any intelligent computer system based on the Von Neumann architecture, will be based on a logic proof system. There will be sentences that can be understood to be true by Humans, but not by the computer system, in particular, the *Gödel sentence* of its proof system. Hence, we are smarter than computers.

A strong advocate of this position was Roger Penrose, Nobel prize Physics, in his book *"The Emperor's New Mind" (1990).*

To explain the superiority of the human brain, he (and others) suggested that there are macroscopic quantum phenomena connected to the human brain's neural activity.

The thesis is considered erroneous by experts (arguments focus on the way Penrose interprets Gödels theorem and are "merciless technical").

6.4 Implications for "deductive logic"

In the view of FO as the logic of deductive reasoning, the results of this section lead to the following uneasy situation. On the one hand, FO lacks expressivity to express certain critical kinds of human domain knowledge. If we decrease the logic in expressivity, the problem will get worse. On the other hand, even now deductive reasoning in FO is not decidable, let alone tractable/efficient. If we extend FO to make it more expressive, this "problem" will only get worse.

This is an instance of the *expressivity / efficiency trade-off*. This trade-off had a strong influence on AI and computational logic. In AI and computational logic, many concluded that FO's

undecidability makes FO unpractical for building effective software solutions for computational problems. Different research directions sprang out from this.

Before we discuss them, let us reflect about the nature of the trade-off. The trade-off does not say that problems in more expressive languages will be solved less efficiently but rather that the class of problems that can be stated in more expressive languages is larger; hence the worst case gets more difficult. Hence, while the increased complexity sounds as a disadvantage for more expressive languages, it is in fact an advantage.

On the other hand, common sense suggests that human software engineers can exploit the expressivity limitations of a less expressive logic $\mathcal{L}_1$ to build more efficient solvers than those for a more expressive logic $\mathcal{L}_2$. Often this is indeed the case. And sometimes the inverse is the case. Sometimes, the particularities of $\mathcal{L}_1$ make it harder to develop good algorithms. Sometimes, the collected efforts of computer scientists developing algorithms for the more general and hence, more interesting inference problem for $\mathcal{L}_2$ lead to systems better than those of $\mathcal{L}_1$. Sometimes these systems, for instances of the inference problem, are better than what can be achieved with "reasonable" effort by hand-made programming. Domains where this arises are, e.g., databases, constraint programming, semantic web languages.

Now, let us return to the research directions that sprung from the expressivity/efficiency trade-off.

One direction was to continue building theorem provers for FO and hope that in practice it will not be so bad. This is what happens in the field of Automated Theorem Proving (ATP) discussed in Section 6.1. This is less unreasonable as it may seem and the ATP community has developed strong theorem provers that at least in certain domains have good use. Thus, ATP is a small but highly specialized community that goes on developing theorem provers for FO, and these solvers are increasingly effective.

Another direction was to restrict FO to a subformalism for which deductive reasoning is decidable or even tractable. This road was taken for instance in the area of *description logics*. Description logics form the basis for the *Web Ontology Languages* like OWL and RDF which are used in Semantic Web applications. The original description logics were small fragments of FO for which deductive reasoning was tractable. A different approach in a similar vein was to restrict FO to a subformalism and develop a proof method such that the human programmer had control over the execution of the proof method. This was implemented in *logic programming (Prolog)*. Also the Prolog language originated by limiting the expressivity of FO, namely to Horn clause logic. Deductive inference in Horn clause logic was still undecidable but Prologs procedural semantics gave human users a way to understand the behaviour of programs, and hence to predict and control it. Of course, in both cases by restricting expressivity, we increase the expressivity problems. In the past two decennia, solvers were developed for gradually more expressive description logics. They are supported by systems such as RACER which, perhaps surprisingly, show tractable behaviour for the early (tractable) description logics and often outperform solvers specifically developed for these low end logics.

Yet another direction was to develop other, cheaper but practically useful forms of inference. In this setting, it is possible to increase the expressivity of FO. This can be seen to be the case, e.g., in Databases and constraint solving. This is the topic of the next Chapter.

6.5 Important for exam

```
big questions:
- compactness theorem and proof
- inexpressivity theorem and proof
- inexpressivity of finite universe, reachability, or of DCA: theorem and proof
- or a combination (e.g., inexpressivity theory plus application for reachability)
In the case of the last sort of question, you should be able to show
the axiomatisation of the negated concept.
Bonus points could be earned if you can solve a variant problem.

For small questions:
- explain undecidability and semi-decidability of validity checking
- explain Goedels incompleteness theorem and its conditions
- explain why it entails undecidability of natural numbers
- explain why HO en FO(.) have no sound and complete proof system
- evaluation of FO from the information centered view (separating
  information from inference)

What I think you need to know of this section to be able to answer these questions:

Section "Deductive inference":

- Understanding of what deduction is, what proof systems and formal
  proofs are, soundness and completeness.

- Knowledge of Compactness theorem; proof.

Section "Some expressivity limitations of FO"

-Knowledge of:
 + Inexpressivity theorem; proof
 + Inexpressivity of finiteness, reachability and DCA ;  proofs
(additional knowledge may lead to bonus points)

Section Undecidability and G\"odels incompleteness theorem

- Understanding of concepts of decidable, recursively enumerable, semi-decidable.

-Knowledge of the main theorems:
 + Undecidability and semi-decidability of Validity/satisfiability/deduction  for FO.
 + G\"odels incompleteness theorem
 + its two  main corollaries:
  - undecidability of the natural numbers
  - lack of sound and complete proof systems for FO(.) and SO
```

Chapter 7

Inference and the FO(.)-KB project

Assume that we built a logic theory of the application domain that contains essential information relevant to certain computational tasks. How can this theory be used to solve the computational tasks that arise in this application domain? This is done by applying one or more suitable forms of inference on the theory.

This is the topic of this chapter. As such this chapter is about the relation between knowledge of an application domain and computational tasks that arise in that domain. The chapter discusses the FO(.)-KB project, the research project of the Knowledge Representation and Reasoning group (KRR) at KU Leuven. We will see the link with various declarative programming paradigms. The chapter is derived from an invited talk at JELIA, the European Conference on Logics in Artificial Intelligence in 2016.

7.1 Motivation

What is the most important human resource after air, water, food? Probably knowledge. Humans experts possess large amounts of (declarative) knowledge. They use it to accomplish tasks and to solve problems. How does this work? This is the basic research question of the field of Knowledge Representation and Reasoning (KRR). If we ever want to be able to build software systems in a principled way, we will need to understand this. This places KRR at the foundations of computer science.

About every area in Computational logic is involved in aspects of this question. However, scientific understanding is partial and scattered over the many fields of computational logic and declarative problem solving.

One issue that fragments computational logic more than anything else is the reasoning/inference task. In the current state of the art, most logics in the field of computational logic are entangled with a specific form of inference:

- Classical first order logic (FO): *deduction*
- Deductive Databases (SQL, Datalog): *query answering*
- Logic Programming: *program execution*
- Answer set Programming (ASP): *answer set computation*
- Inductive Logic Programming: *inductive inference*
- Abductive Logic Programming: *abductive inference*
- Constraint Programming (CP): *constraint solving*

- Temporal logics: *model checking*
- Description logics: *subsumption inference*
- Planning language PDDL: *planning inference*
- ...

Thus, for every form of inference a new logic is developed. From the perspective of logic as the study of inference, this is not unnatural. However, from the perspective of logic as a language to express knowledge, the entanglement of logic and inference is unfortunate. I argue this with two cases.

Consider the following proposition: *Every lecturer teaches at least one course in the first bachelor.*

$$\forall x(Lecturer(x) \Rightarrow \exists c(Course(c, 1stbach) \wedge teaches(x, c)))$$

What is its purpose? What task is it to be used for? It could be a query to a database; or a constraint in a course assignment problem; it could be a desired property, to be verified from a formal specification of the course assignment domain, etc. A declarative proposition has no inherent purpose or task. The proposition could be relevant to many different sorts of tasks. In the current state of the art, a different logic is needed to represent this proposition depending on the task to be solved. This is not right.

For a second case, consider the graph coloring problem. What is the central "knowledge" in the graph coloring problem? That no two adjacent vertices have the same color. In FO, this can be expressed as:

$$\forall x \forall y(G(x, y) \Rightarrow Col(x) \neq Col(y))$$

This is a natural representation of the central piece of knowledge in this problem. How to solve a specific graph coloring problem in FO using this proposition? This question amounts to what kind of inference is needed to apply to this formula to solve this problem. The fact is that until fairly recently, for FO, this question was not even posed. FO was seen as the logic of **deductive reasoning** and continues to be seen this way by many to this day. Deduction however is utterly useless for solving a graph coloring problem. Therefore, FO seems useless for this problem. Instead, new logics equipped with the suitable form of inference were developed to handle problems like this: e.g., Constraint Programming Languages such as Ilog, Zinc, CLP, ... But in fact, the form of inference to solve the graph coloring problem using the FO specification is *model generation/expansion*.

From a scientific point of view, this situation is deeply unsatisfactory. Is declarative knowledge not supposed to be independent of the task (and hence, of a specific form of inference)? Should it not be possible to solve multiple types of tasks using information expressed in the same language?

The FO(.)-KB project: an outline The FO(.)-KB project is the research project in the Knowledge Representation & Reasoning (KRR) research group of KU Leuven. In this approach, a declarative logic theory is not a program, it cannot be executed, it is not even a description of a problem. It is a bag of (descriptive) information.

On the logical level, the goal is study "knowledge" by a principled development of expressive KR languages. The goal is to develop KR logics with a clear informal semantics and a formal model semantics that formalizes this informal semantics. The logic is expressive enough such that domain knowledge relevant to solving the problem *can* be expressed in a compact, modular way. Classical first-order logic (FO) is taken as foundation but extended and improved where necessary.

On the application level, the goal is to develop a typology of tasks and computational problems in terms of (the same) logic and inference. We are eagerly searching for novel ways of using declarative specifications to solve problems. We are also searching for applications where different forms of inference on the knowledge base are required to obtain the desired functionality. Such applications show a form of *reuse* of the knowledge base that is impossible to achieve in systems that support only one inference task.

On the inference level, the goal is to build solvers for various forms of inference for FO(·). To this aim, we are integrating various solving techniques from various declarative programming paradigms in the Knowledge Base System IDP. IDP is the laboratory for our scientific research.

7.2 Why FO as a foundation?

FO is the outcome of 18's and 19's century's research in "laws of thought". E.g., Leibniz , De Morgan, Boole, Frege. FO offers a small set of connectives:

$$\wedge, \vee, \neg, \forall, \exists, \Leftrightarrow, \Rightarrow$$

These are essential for KR and are correctly formalized in FO. As a consequence, every expressive declarative modelling language should be expected to have a substantial overlap with FO although syntactic sugar, conceptology and terminology may sometimes obscure the relationship, e.g., SQL, constraint languages, Alloy, ...

Another crucial aspect of FO for KR is its mathematically precise informal semantics. E.g., $\forall x(Human(x) \Rightarrow Man(x) \vee Woman(x))$ means *All humans are men or women.*

On the other hand, FO is not expressive enough for practical KR. To turn FO into a practical KR language, it needs to be extended. In the project, several extensions are studied an integrated in FO:

$$\text{FO(Types,ID,Agg,Arit,Causation,...)}$$

We call this the FO(·) language framework.

The expressivity/efficiency trade-off reconsidered Deductive inference in FO is undecidable but still semi-decidable. In extensions, e.g., with second order quantification or with inductive definitions, the situation is worse. How to deal with this?

It is in the nature of knowledge that the same knowledge can be useful or relevant in a variety of problems. The computational complexity of these problems is varying widely. Some of these problems are "easy", e.g., checking truth of a proposition in a structure; other computational problems are difficult or even unsolvable in full generality, e.g., proving a proposition from a theory for verification. This is the nature of knowledge. In a knowledge-centered approach such as the FO(·)-KB project, one has to accept that some sort of problems are untractable or even undecidable, while other problems are solvable. It may well be that in practical applications, the required forms of inference are mostly of the easy kind. In the FO(·)-KB project, the focus is mostly on problems that can be solved within a finite domain.

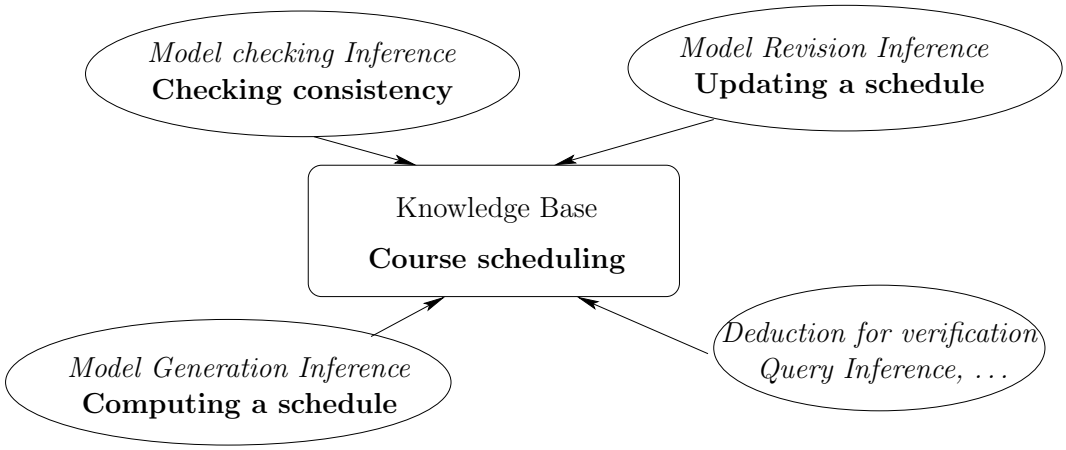


Figure 7.1: The knowledge base approach to course scheduling

7.3 The knowledge base paradigm

A knowledge base system is a system that manages a knowledge base and supports multiple forms of inference on it. The concept is illustrated in Figure 7.1 in the application domain of university course scheduling.

Knowledge in this domain includes properties such as:
- all lectures take place in a room and at some time point;
- lecturers and students cannot attend different lectures simultaneously;
- rooms cannot host more than one lecture at the same time;
- the capacity of rooms should be sufficient to hold student groups.
- ... In practice, many more constraints arise in such domains.

The knowledge base T_s consists of a theory representing this knowledge.

At several stages during the academic year, different tasks arise that can be solved by applying the appropriate form of inference on the knowledge base. Recall the definition of inference from Definition 1.3.2. It is a computational problem with as input a proposition or theory and its output is invariant under substituting the input formula or theory by a logically equivalent one.

In Chapter II, we defined different inference problems for arbitrary logic with a satisfaction relation $\models$: evaluation, model checking, satisfiability checking, model generation, optimization, deduction and possible values problems. Here, we reintroduce these forms of inference and some extra forms of inference in the context of FO(.) and illustrate their application.

Some common tasks that arise in a university scheduling application are the following:

- *Verification*: does T_s entail a proposition φ?

  Deduction:
  input: T_s, φ
  output:$(T_s \models \varphi)$

- Compute a schedule; a model of T_s comprises a value for predicates $RoomOf, TimeOf$ representing the schedule.

> Model expansion:
> input: $T_s, \mathfrak{A}_i$ with $\mathfrak{A}_i$ a structure representing data.
> output: a model $\mathfrak{A}$ of T_s expanding $\mathfrak{A}_i$, or UNSAT

Typically such a problem is subject to some optimality criterion. In that case, a variant of this form of inference is needed.

- Check if a manually modified schedule $\mathfrak{A}'$ is still correct.

> Model checking:
> input: the modified schedule $\mathfrak{A}'$, T_s;
> output: $(\mathfrak{A}' \models T_s)$

- Compute answer to query φ or $\{x \mid \varphi[x]\}$ in schedule $\mathfrak{A}$.

> Query answering:
> input: $\mathfrak{A}, \varphi$ or $\{x : \varphi[x]\}$;
> output: $(\mathfrak{A} \models \varphi)$ or $\{x : \varphi[x]\}^{\mathfrak{A}}$.

- Modify given schedule $\mathfrak{A}$ to satisfy a new proposition φ.

> Model revision:
> Input: $\mathfrak{A}, \varphi, T_s$;
> output: structure $\mathfrak{A}'$ close to $\mathfrak{A}$ such that $\mathfrak{A}' \models T_s \wedge \varphi$.

In the following section, we discuss some applications.

7.4 A KB-solution for Interactive Configuration

In many configuration problems, a user is constructing a configuration under complex constraints. E.g., configuring your study program, a schedule for a hospital or university, a computer in an on-line shop, a car, a loan in a bank, a mechanical device, a plan of a building, etc.

Building good configuration software to support the user is difficult. The user chooses, the system supports. During the interaction, a partial configuration is constructed, extended, revised according to user's wishes. The user is not or only partially aware of the constraints. The system is to support the user in his choices and to take care for maintaining the consistency of the solution.

Industry is aware that good solutions for interactive configuration are extremely difficult to build with standard software technology. Current technologies fall short: spreadsheets, business rule systems, programming languages. Why? The specifications of correct configurations are complex and tend to change rapidly. The range of needed functionalities is high. Snippets of code concerned with some specific constraint spread out at many places in the software, showing a lack of reuse and leading to difficult maintenance.

A solution based on the KB-paradigm The idea is to express the laws of correct configurations in a knowledge base, and to provide different functionalities of the system by various forms of inference. In an interaction between user and system, a partial configuration is built and gradually extended. All knowledge is maintained in one knowledge base; the current partial configuration is modelled as a partial structure. The system performs various forms of inference on the knowledge base and the partial structure to support the user.

The idea is illustrated with a demo for the interactive configuration of course selection at a fictitious university. This demo is derived from a challenge problem developed by the VUB (Free University of Brussels) as a test for software companies and methodologies. The demo is implemented with the IDP system. It is available at:

`http://krr.bitbucket.io/courses/`

The knowledge base T_c contains a number of laws such as:
- compulsory courses have to be selected;
- a number of optional courses have to be selected;
- one out of three possible modules have to be selected;
- each module has a number of module courses which are to be followed exactly if the module is selected.
- etc.

Models of this theory satisfy all laws. The theory can be inspected by clicking the button *Edit IDP source*.

The application is built as a shallow GUI built in json which communicates information back and forth between the knowledge base system and the user. The GUI is presented in Figure 7.2. The KB system maintains a partial structure $\mathcal{I}$ representing the current choices of the user and the implications that these choices entail. The system supports the users in various ways, using 5 different forms of inference. Below, we describe them.

Definition 7.4.1. A partial structure $\mathcal{I}$ consists of a domain $D_{\mathcal{I}}$, function values for function symbols and *partial sets* for predicate symbols. A partial set is a function from tuples to $\{\mathbf{t}, \mathbf{f}, \mathbf{u}\}$ where $\mathbf{u}$ is called *undefined*.

On partial structures, the *precision relation* is defined: $\mathcal{I} \leq_p \mathcal{I}'$ if $\mathcal{I}, \mathcal{I}'$ have the same domain and value of function symbols and, if $P^{\mathcal{I}}(\bar{d}) \neq \mathbf{u}$ then $P^{\mathcal{I}}(\bar{d}) = P^{\mathcal{I}'}(\bar{d})$.

In words, the partial structure $\mathcal{I}$ is less precise than $\mathcal{I}'$ (notation $\mathcal{I} \leq_p \mathcal{I}'$) if the interpretations of predicates in $\mathcal{I}'$ extend those of $\mathcal{I}$. That is, if any atom $P(\bar{d})$ has a defined value in $\mathcal{I}$ (i.e., $P^{\mathcal{I}}(\bar{d}) \neq \mathbf{u}$) then $P(\bar{d})$ has a defined value in $\mathcal{I}'$ as well and $\mathcal{I}$ and $\mathcal{I}'$ agree on its value (i.e., $P^{\mathcal{I}}(\bar{d}) = P^{\mathcal{I}'}(\bar{d})$).

We say that a standard two-valued structure $\mathfrak{A}$ expands a partial structure $\mathcal{I}$ if $\mathcal{I} \leq_p \mathfrak{A}$.

Functionalities: inferences on $\mathcal{I}$ and T_c

1. *Propagation inference*

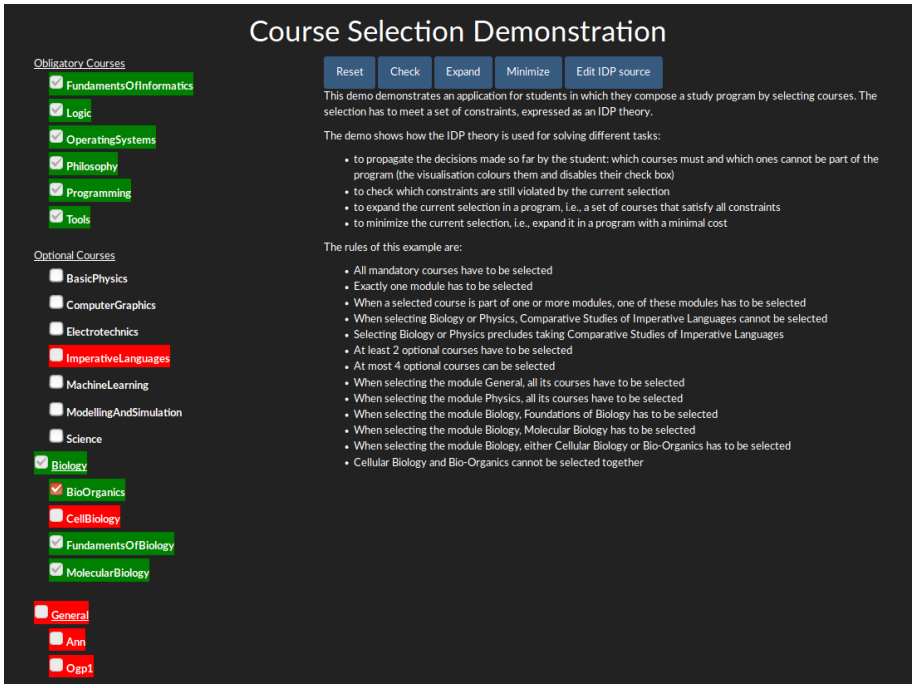


Figure 7.2: Course selection demo

> Propagation Inference:
> Input: $T_c, \mathcal{I}$
> Output: $\mathcal{I}'$ where $\mathcal{I} \leq_p \mathcal{I}'$ and for every model $\mathfrak{A}$ of T_c such that $\mathcal{I} \leq_p \mathfrak{A}$, it holds that $\mathcal{I}' \leq_p \mathfrak{A}$.

In the demo, propagation inference is called initially and whenever the user makes a choice. The effect is that the implications of his choices under T_c are propagated. Forced choices are colored green, forbidden choices red.

A special form of this inference is *optimal propagation*, where the result $\mathcal{I}'$ is the most precise partial structure such that every model $\mathfrak{A}$ of T_c expanding $\mathcal{I}$ also expands $\mathcal{I}'$.

Stated differently, $\mathcal{I}'$ is obtained from $\mathcal{I}$ as follows:

- an unknown atom A of $\mathcal{I}$ that is true in all models $\mathfrak{A}$ expanding $\mathcal{I}$, is true in $\mathcal{I}'$
- an unknown atom A of $\mathcal{I}$ that is false in all models $\mathfrak{A}$ expanding $\mathcal{I}$, is false in $\mathcal{I}'$.

2. *Model checking inference.* This form of inference was defined before. In the demo, it is called when clicking button *Check*. First, the system transforms the current partial structure $\mathcal{I}$ into a total structure $\mathfrak{A}$ by turning every *undefined* atom into *false*. Then *model checking inference* is applied on $\mathfrak{A}$ and T_c to computed whether $\mathfrak{A} \models T_c$.

3. *Explanation inference.* At several times, explanations are generated. An explanation is a logical argument why an axiom or group of axioms is violated, why a certain choice is forced or impossible. A simpler form is implemented in this demo: it reports which axioms are violated in the partial structure $\mathcal{I}$ or the structure $\mathfrak{A}$ computed in the model checking step.

4. *Model expansion inference.*

> Model expansion:
> Input: $\mathcal{I}, T_c$
> Output: a model $\mathfrak{A}$ of T_c such that $\mathcal{I} \leq_p \mathfrak{A}$.

It serves to complete the current partial structure to a model of the theory. This is useful to fill in the "don't cares" of the student. This form of inference is called by pushing the button *Expand*.

5. *Optimisation inference.*

> Optimisation:
> Input: $\mathcal{I}, T_c, t$ where t is a numerical term
> Output: a model $\mathfrak{A}$ of T_c such that $\mathcal{I} \leq_p \mathfrak{A}$ and $t^{\mathfrak{A}}$ is minimal.

This inference is called when clicking button *Minimize*. In this application, the term t expresses the study load of the selected program. Hence, a model with minimal studyload is computed. In the theory, the term is declared as the IDP term `optTerm`. It is the following expression counting the total value of selected courses:

```
sum{x[Cost] c[Course]:  SelectedCourse(c) & HasValue(c)=x :  x}
```

Discussion This sort of application is a show case for the KB-paradigm.

The system uses a formal specification to run a software tool. The application shows a perfect *separation* of domain knowledge versus computational problems. As a consequence, changing the specification T_c immediately affects the behaviour all components of the system. One can test this by editing the IDP source. [1]

The application shows a strong **reuse** of the knowledge base. This is a form of reuse that cannot (easily) be achieved with standard programming languages or uni-inferential declarative programming languages.

Publication:

- Pieter Van Hertum, Ingmar Dasseville, Gerda Janssens, Marc Denecker: The KB Paradigm and Its Application to Interactive Configuration. PADL 2016.

Interactive Decision Enactment `http://krr.bitbucket.org/autoconfig/`

Another application of the same idea was for interactive configuration of a car insurance in Figure 7.3. This application is a benchmark problem in the area of Business Applications. In this system, a simple GUI for the configuration problem is automatically generated from the vocabulary. By adding new symbols to the vocabulary, the GUI is extended automatically with new fields. This application won the RuleML 2016 challenge.

Publication:

[1] E.g., replace $\forall$ `x:  MandatoryCourse(x)` $\Rightarrow$ `SelectedCourse(x)` in axiom 1 of theory T by `true` and save.

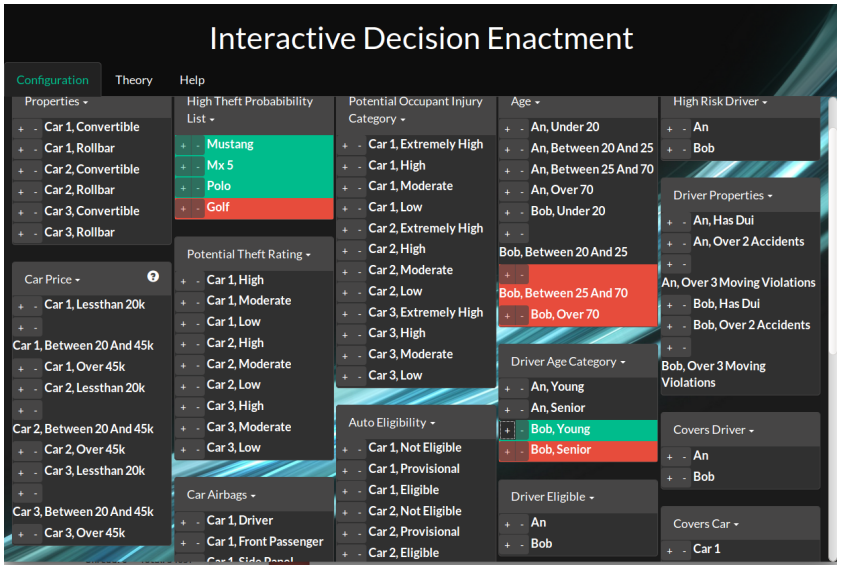


Figure 7.3: Car Insurance Decision Enactment

- Dasseville, Ingmar; Janssens, Laurent; Janssens, Gerda; Vanthienen, Jan; Denecker, Marc. Combining DMN and the knowledge base paradigm for flexible decision enactment, RuleML, July 2016.

7.5 Inference with definitions

Given a definition Δ over Σ with parameter symbols $Param(\Delta)$ and defined symbols $Def(\Delta) = \Sigma \setminus Param(\Delta)$, a special form of model expansion is the following:

> Δ-model expansion:
> Input: a $Param(\Delta)$-structure $\mathfrak{A}_p$
> Output: the structure $\mathfrak{A}$ satisfying Δ expanding $\mathfrak{A}_p$

I.e., compute the unique model of Δ expanding $\mathfrak{A}_p$. This is an instance of the model expansion problem. It is an interesting subproblem. It is deterministic, i.e., there is a unique solution. It has many applications. Moreover, efficient special purpose algorithms exist.

In the IDP3 sytem, this is implemented by the procedure `calculatedefinitions(T,S)`. For an example see `https://dtai.cs.kuleuven.be/software/idp/examples`, select the N-queens example.

Another form of inference specific to definitions is Δ-*model revision*. The context is that the unique model $\mathfrak{A}$ of definition Δ for $Param(\Delta)$-structure $\mathfrak{A}_p$ has been computed. Now, the parameter structure $\mathfrak{A}_p$ is updated, by adding or deleting some tuples to the value of predicates in $\mathfrak{A}_p$. The challenge of model revision is to update $\mathfrak{A}$ in an *incremental way*. Indeed, small updates of $\mathfrak{A}_p$ typically cause small updates of $\mathfrak{A}$. Rather than recomputing $\mathfrak{A}$ from scratch, propagate the given update to modify $\mathfrak{A}$.

Hidden in different syntactical forms, various software systems contain definitions and implement the above forms of inference.

In Databases

- Views in databases $\sim$ defined symbols
- View definitions $\sim$ FO(ID) definitions
- *view materialization* $\sim$ Δ-model generation
- *incremental materialized view update* $\sim$ Δ-model revision

In spreadsheets Assume that a field or a column or row A is defined in terms of other fields or columns or rows B by some symbolic expression. When a spreadsheet computes A from B, it can be seen to perform a form of Δ-model generation. To update A after an update of B is Δ-model revision.

Intermezzo 7.5.1. IDPd3 Δ-model expansion is used in *IDPd3*, the system used in IDP for computing a visualisation of structures. An example is found here:

`dtai.cs.kuleuven.be/krr/idp-ide/?present=SudokuVisualisatie`

This application serves to solve sudoku puzzles and to visualise the outcome.

The theory `T` expresses the 3 laws of sudokus: one occurrence of each number in each row, column and block. Model expansion inference solves a puzzle specified in the input structure by returning a model $\mathfrak{A}$ in which $sudoku^{\mathfrak{A}}$ represents the solution.

To visualize this solution, the user theory `T_D3` below is used. It is a definition of *IDPd3 graphical predicate and function symbols*. These predicates are defined in terms of symbols interpreted in $\mathfrak{A}$. Δ-model expansion on `T_D3` and input structure $\mathfrak{A}$ computes a value for the graphical predicates which is then transferred to the graphical program **d3**.

The operation of IDPd3 illustrated:

Input structure

$$sudoku = \{1,1 \to 1; \dots; 9,9 \to 4\}$$
$$\dots$$

$\downarrow$ Δ-model generation on T_D3

$$d3\_type = \{1, Cell(1,1) \to rect; 1, Text(1,1) \to 1; \dots\}$$
$$d3\_color = \{1, Cell(1,1) \to white; 1, Text(1,1) \to \text{"}black\text{"}; \dots\}$$
$$d3\_x = \{1, Cell(1,1) \to 5; 1, Cell(1,2) \to 10; \dots\}$$
$$\dots$$

$\downarrow$ translation to d3 input + d3

The program d3 visualises this input.

```
theory T_D3 :  V_out {
  {
    d3_type(1, Cell(r,k)) = rect <-.
    d3_rect_width(1, Cell(r,k)) = 4 <-.
    d3_rect_height(1, Cell(r,k)) = 4 <-.
    d3_color(1, Cell(r,k)) = "white" <-.
    d3_x(1, Cell(r,k)) = 5*k <-.
    d3_y(1, Cell(r,k)) = 5*r <-.

    d3_type(1, Text(r, k)) = text <-.
    d3_x(1, Text(r, k)) = 5*k <-.
    d3_y(1, Text(r, k)) = 5*r + 1 <-.
    d3_text_size(1, Text(r, k)) = 3 <-.
    d3_text_label(1, Text(r, k)) = t <-
      sudoku(r, k) = c & toString(c) = t.
    d3_color(1, Text(r, k)) = "black" <-.

    d3_order(1, Cell(r, k)) = 0 <-.
    d3_order(1, Text(r,k)) = 1 <-.
  }
}
```

This definition defines slide 1 (argument 1) with:

- for each cell (r, c) a rectangular object denoted Cell(r,c), and a text object Text(r,k).
- It defines type, width, height, color, x and y position of the rectangular object.
- It defines type, text size and label, color, x and y position of the text object
- The `sudoku` number at cell (r, c) is the label of the text object..
- that text objects are in front of rectangular objects

Δ-model expansion expands a structure interpreting `sudoku` into a structure interpreting all these graphical symbols. This is fed into **d3**.

For more illustrations, see `dtai.cs.kuleuven.be/krr/idp-ide`, select "File", "9.Visualisations".

7.6 Imperative + Declarative Programming in IDP3

The IDP3 system supports a prototype knowledge-based programming environment in Lua. The environment includes high level logical objects such as vocabularies, theories, structures. It provides functionalities for manipulation and inference of these integrated and implemented in the language Lua. This supports a novel way of mixing declarative and procedural programming.

This is demonstrated in the following demo for generating Sudoku-puzzles. A requirement for a sudoku-puzzle is that it has a unique solution. Another condition that is related to its difficulty is that it is minimal. I.e., if a value of the puzzle is deleted, the puzzle has strictly more than one solutions.

In this demo, minimal Sudoku puzzles are generated.

Background knowledge base in IDP

```
vocabulary sudokuVoc {
    extern vocabulary grid::simpleGridVoc
    type Num isa nat
```

```
    type Block isa nat
    Sudoku(Row,Col) :  Num
    InBlock(Block,Row,Col)
}
theory sudokuTheory :  sudokuVoc {
    !  r n :  ?1 c :  Sudoku(r,c) = n.
    !  c n :  ?1 r :  Sudoku(r,c) = n.
    !  b n :  ?1 r c :  InBlock(b,r,c) & Sudoku(r,c) = n.
    !  b r c :  InBlock(b,r,c) ⇔ b = ((r-1)/3)*3 + ((c-1)/3) + 1.
}
```

Procedure to generate minimal Sudoku-puzzles

```
Puzzle := empty
Generate at most 2 solutions for Puzzle
While 2 solutions were found do{
    Select a random position where the two solutions differ
    Extend Puzzle with the value of the first solution at this position
    Generate at most 2 solutions for Puzzle
    }
For each position of Puzzle that contains a value do {
    Delete the value at this position
    Generate at most 2 solutions for Puzzle
    If there are two solutions, undo the deletion of the value.
    }
Visualize the puzzle and its unique solution
```

The Lua implementation of part of this procedure is in Figure 7.4.

This application uses two sorts of inferences:**model expansion** for generating solutions to puzzles, and **Δ-model expansion** for visualizing puzzles and their solution.

7.7 Various forms of inference in IDP

Queries in IDP The IDP procedure `Query(Q,S)` implements query inference and computes the value of the set expression contained in a query term `Q` in the context of structure `S`. A query declaration associates a symbolic name to a set expression. An example is found in:

`http://dtai.cs.kuleuven.be/krr/idp-ide/?present=MapColoringMXQuery`

It contains the following query declaration:

```
query Q:V { {x[Color]: ?y[Area]:Coloring(y)=x} }
```

Model expansion and optimisation inference Model expansion inference in IDP is performed by several procedures including `modelexpand(T,S)` which returns zero, one or more models of `T` expanding `S` (the maximum number can be declared via an option).

```
vprocedure createSudoku() {
    math.randomseed(os.time())
    local puzzle = grid::makeEmptyGrid(9)

    stdoptions.nrmodels = 2
    local currsols = modelExpand(sudokuTheory,puzzle)

    while #currsols > 1 do{
        repeat
            col = math.random(1,9)
            row = math.random(1,9)
            num = currsols[1][sudokuVoc::Sudoku](row,col)
            until num ~= currsols[2][sudokuVoc::Sudoku](row,col)

            makeTrue(puzzle[sudokuVoc::Sudoku].graph,row,col,num)
            currsols = modelExpand(sudokuTheory,puzzle)
        end

        printSudoku(puzzle)
    }
```

Figure 7.4: Lua procedure to generate sudoku puzzles in IDP3

The procedure `allmodels(T,S)` computes all model expansions.

Another variant is the procedure `minimize(T,S,t)` which performs optimisation inference. It computes one or more models of `T` expanding `S` that minimize the value of the term `t`.

These forms of inference are illustrated in:

`http://dtai.cs.kuleuven.be/krr/idp-ide/?present=MapColoringMXOpt`

These procedures may take as input a structure `S` that is a standard structure of a subvocabulary. However, the output may also be a partial structure. An illustration of how to express a partial structure is found in the Einstein puzzle at:

`https://dtai.cs.kuleuven.be/software/idp/examples`

It contains a declaration of the form:

```
structure S:V {
   ...
   OwnsHouse<ct> = {English->Red; }
   ...
}
```

`OwnsHouse` is declared as a function symbol. The above assignment specifies that this function is all unknown except that `OwnsHouse(English)=Red`. The suffix `<ct>` indicates that this assigment specifies certainly true values. It is also possible to specify certainly false values, e.g.,

`OwnsHouse<cf> = {Japanese->Blue; }`

Model expansion is the logical variant of *Constraint Programming*, the declarative programming paradigm for solving search problems. Many constraint programming engines exist such as ILOG, Zinc, etc. An example application of constraint solving/model expansion is the 2 dimensional packing problem where the goal is to put a number of squares of different size on a rectangular grid.

`http://dtai.cs.kuleuven.be/krr/idp-ide/?src=31c913a78b077bf6ab89`

A well-known optimisation problem is the knapsack problem. The goal is to select a set of objects, each with a value and weight, to put them in a knapsack with a maximal weight. The goal is to optimize the total value of the knapsack. A solution in IDP is found at:

`http://dtai.cs.kuleuven.be/krr/idp-ide/?src=ace6dc004bfeff3dcca3`

Propagation inference Propagation inference was introduced in the discussion of the course selection application. Propagation inference is useful to compute possible values of symbols, given a partial structure and a theory. IDP supports different forms of propagation inference: optimal propagation (which is expensive) but also (cheap but incomplete) symbolic and non-symbolic propagation. The procedures are `propagate(T,S)`, `groundpropagate(T,S)` and `optimalpropagate(T,S)`

For a small problem illustrating these three forms of propagation, select the last example ("Propagation & Predicate Table Access") on `https://dtai.cs.kuleuven.be/software/idp/examples`.

7.8 Reasoning on LTC theories

In the context of Linear Time Calculus, a range of different extra forms of inference are possible:

- Model expansion and optimisation inference can be used for planning (with optimisation).
- `initialise(T,S)`, `progress(T,S)`: to compute a set of initial states of a LTC `T` expanding structure `S`, respectively, to compute a set of progressions of `T` from state `S`. These forms of inference can be used for building interactive simulation as demonstrated in a pacman demo. `https://dtai.cs.kuleuven.be/krr/idp-ide/?chapter=intro/9-IDPD3`.
- `isinvariant(T,Inv,S)`: proving that `Inv` is an invariant of LTC `T` in the context of models expanding structure `S`.

To the best of our knowledge, IDP is the only system that can use the same formal LTC both to simulate an LTC theory as well as compute (optimal) plans from it.

The company LogicBlox uses progression as a basic inference step to build large business applications. `http://www.logicblox.com/`

7.9 Conclusion

Deductive inference (theorem proving) was the original form of inference studied for logic. It is important for *verification*: verifying that a system satisfies correctness conditions *under all*

circumstances. It is a computationally complex form of inference, as is attested by its undecidability in a simple language such as FO. Many common tasks in software products can be achieved by other forms of inference: model checking, query answering, finite model generation, model revision .... The good news is: each of these forms of inference is *computationally cheaper*, and *practically more often useful* than theorem proving. Deduction is for the design phase (verification); the cheaper forms of inference are for the production phase.

One question that arises is to what extent software systems can be developed by application of logic inference methods on formal specification and what would be the benefit. Of course, the challenges at the computational level remain huge. But where this approach works, there is little doubt that great benefits are feasible on the level of standard software metrics:

Compactness – correctness – separation of concerns – reuse – maintainability

The company LogicBlox argues that by using their system, software development is fastened by a factor 50× compared to standard software methodologies. Also we (in KRR) have seen sometimes enormous improvements (one time, replacing a program by a logic theory that was 400 times smaller). Another example dating from a few years back was the system Anton for music generation by *model generation* on a variant of FO(.). The authors claimed that their 500 lines of logic specification performed similar as a program of 88.000 lines of C code.

In a growing range of applications, logical methods become more and more effective. One area for which I believe it is fair to say that logic and inference have found solid ground in industrial applications is database technology. Although many do not longer seem to view database technology as applied logic. Another area where logic gains ground is in Constraint solving.

From a different point of view, I claim that the logic perspective shows overlaps and correspondences in areas of declarative programming – overlaps in the languages, in the inference implemented by the tools, in the techniques and algorithms used to implement this inference. An enormous amount of technologies have been developed in different areas: database technologies, Ontology language technologies, Logic Programming, Constraint solving, SAT, Answer Set Programming, domain specific configuration languages and solvers, Expert system and Business Rules technologies, ... The challenge is to integrate and generalize the many existing technologies in generic inference systems for logic.

The specific contribution of the KRR group is to develop applications by applying different inference tools to FO(.) theories. In this respect, we belong to the pioneers. At present, we are searching for industrial applications for IDP, especially in areas of knowledge-intense applications where multiple forms of inference are of use. The group continues to improve its systems by integrating existing technologies in our systems. You can help us – do a thesis in our group.

7.10 Important for the exam

```
Inference:

-Understanding of the principle of solving problems by applying
  inference to specifications.
-Definitions of the most important forms of inference, understanding of
```

their utility.
-Understanding of the knowledge base paradigm as illustrated in interactive configuration.

Large question: give
an overview of inference problems or of the knowledge base
paradigm. You could study this by making a list of inference
problems, their formal definitions, a few examples, other fields
where this form of inference is applied.
You could organize your answer on the interactive
configuration problem that illustrates 5 forms of inference AND
illustrates the separation of information and problem, and
the reuse of a theory.

Chapter 8

Algorithms for finite satisfiability checking in propositional and predicate logic

In this chapter, we see algorithms for satisfiability checking of propositional logic and finite model expansion for predicate logic.

8.1 Proving satisfiability of Propositional Logic

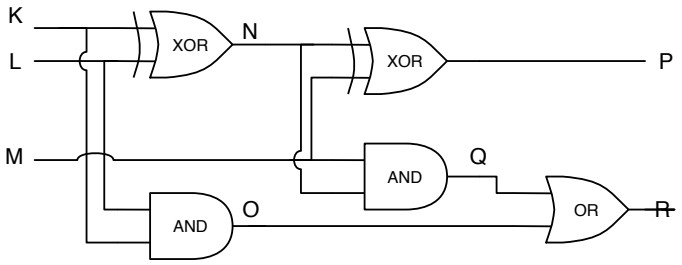
8.1.1 Propositional logic

Propositional logic or propositional calculus (PC) is the subformalism of FO obtained by restricting vocabularies Σ to propositional symbols, predicate symbols of arity 0. Propositional logic does not include quantifiers. The connectives are the standard ones $\neg, \wedge, \neg, \Rightarrow, \Leftrightarrow$. The Σ-structures $\mathfrak{A}$ are boolean functions of Σ.

The history of propositional logic goes back to Boole who in 1854 published his work in a paper *An Investigation of the Laws of Thought, on Which are Founded the Mathematical Theories of Logic and Probabilities*. This work introduced Boolean algebra.

Propositional logic is a primitive modelling language. It is used frequently as a language into which specifications expressed in more expressive languages are translated. It can be considered as the "machine language" of logic.

Example 8.1.1. A boolean circuit example:



This boolean circuit represents 2 formulas

$$(K \,\mathrm{XOR}\, L) \,\mathrm{XOR}\, M \text{ and } (K \wedge L) \vee (M \wedge (K \,\mathrm{XOR}\, L))$$

where XOR is exclusive disjunction. Recall that $(K \,\mathrm{XOR}\, L) \equiv (K \Leftrightarrow \neg L)$.

Example 8.1.2. Some people took drugs. The suspects are: Sam, Michael, Bill, Richard, Matt. There is also Tom but he is innocent.

- Sam said: Michael or Bill took drugs, but not both.
- Michael said: Richard or Sam took drugs, but not both.
- Bill said: Matt or Michael took drugs, but not both.
- Richard said: Bill or Matt took drugs, but not both.
- Matt said: Bill or Richard took drugs, but not both.

Of these 5 statements, 4 are true, one is false.

Tom said: If Richard took drugs, then Bill took drugs. This is true.

Who doped?

We use the following vocabulary. Statements are made by six different people. The intended interpretation of the symbols `ss, mis, bs, rs, mas, ts` is that the statements made by respectively Sam, Michael, Bill, Richard, Matt and Tom are true (`ss` stands for "Sam Said"). The intended interpretation of the symbols `sd, mid, bd, rd, mad` is that respectively Sam, Michael, Bill, Richard, and Matt are doped (`sd` stands for "Sam Doped").

In terms of this vocabulary, the given information can be specified as follows:

```
ss ⇔ (mid ⇔ ¬bd).
mis ⇔ (rd ⇔ ¬sd).
bs ⇔ (mad ⇔ ¬mis).
rs ⇔ (bd ⇔ ¬mad).
mas ⇔ (bd ⇔ ¬rd).
ts ⇔ (rd ⇒ bd).
ts.
(ss ∧ mis ∧ bs ∧ rs ∧ ¬mas)∨
(ss ∧ mis ∧ bs ∧ ¬rs ∧ mas)∨
(ss ∧ mis ∧ ¬bs ∧ rs ∧ mas)∨
(ss ∧ ¬mis ∧ bs ∧ rs ∧ mas)∨
(¬ss ∧ mis ∧ bs ∧ rs ∧ mas).
```

Exercise 8.1.1. *Consider the finite set of binary strings:*

$$\left\{ \begin{array}{l} (000000), (100000), (110000), (111000), (111100), (111110), \\ (111111), (011111), (001111), (000111), (000011), (000001) \end{array} \right\}$$

Represent this set in a compact propositional formula. You may write formulas compactly using indexing as in $\wedge_{0<i<10} b_i$.

Satisfiability Inference In this chapter, we are interested in algorithms implementing *satisfiability checking* and *model generation* for propositional and predicate logic.

Satisfiability checking inference:
- Input: a PC formula φ
- Output: **t** (true) if φ is satisfiable; **f** otherwise

As it turns out, these algorithms do this by computing models of a propositional theory. That is, they can be used to implement also the following form of inference.

Model generation inference:
- Input: a PC formula φ
- Output: one or more models $\mathfrak{A}$ of φ; "UNSAT" if φ is unsatisfiable;

The decision problem of satisfiability checking has several sorts of applications. E.g., when we build a theory of integrity constraints in a domain, e.g., in the context of a database application, or a constraint solving problem, or a piece of legislation (a set of laws), etc., a basic correctness test for the theory is that it is satisfiable. E.g., when we want to verify whether a given specification T entails a desired or expected property φ, we test whether $T \cup \{\neg\varphi\}$ is unsatisfiable.

In many applications, the models of a theory provide useful information. In constraint solving applications, the solution to the constraint problem is (part of) a model of the theory. When a verification that $T \models \varphi$ fails because $T \cup \{\neg\varphi\}$ turns out to be satisfiable, a model of the latter theory represents a possible state of affairs in which T is true and the expected proposition φ is false. By inspecting this model, we can detect the error in our reasoning and correct it — by refining T or φ.

In predicate logic, satisfiability checking is undecidable (and not even semi-decidable). But in propositional logic, the problem is decidable.

Exercise 8.1.2. *Explain that satisfiability checking in FO is not semi-decidable.*

8.1.2 Satisfiability inference in PC

Consider the Algorithm 0.1:

- Exhaustively enumerate all structures $\mathfrak{A}$ interpreting φ and verify $\mathfrak{A} \models \varphi$.

Algorithm 0.1 is underlying the method of *truth tables* as illustrated below:

p	q	(	p	$\Rightarrow$	q	)	$\Leftrightarrow$	(	$\neg$	p	$\vee$	q	)
t	**t**		**t**	**t**	**t**		**t**		**f**	**t**	**t**	**t**	
t	**f**		**t**	**f**	**f**		**t**		**f**	**t**	**f**	**f**	
f	**t**		**f**	**t**	**t**		**t**		**t**	**f**	**t**	**t**	
f	**f**		**f**	**t**	**f**		**t**		**t**	**f**	**t**	**f**	

Each row corresponds to one structure. After one step, the satisfiability of the formula was determined. After the last step, the validity of the formula was determined.

This algorithm is unfeasible for any but the smallest numbers of symbols. With n symbols, 2^n number of structures are to be tested. For $n = 50$, $2^n \approx 10^{15}$, with $n = 60$, $2^n \approx 10^{18}$. In

current applications, CNF formulas with hundred thousands or even million of symbols are no exception.

Complexity The satisfiability problem is a prototypical NP-complete problem (in the size of the formula). This is Stephen Cook's theorem [1971].

Nobody has been able to prove that NP $\neq$ P, i.e., there are no polynomial algorithms to solve such problems. Most people believe this to be the case.

"NP =P" is a millennium prize problem, a problem selected by the Clay Mathematics Institute as one of the seven most important problems in Mathematics. A proof or a proof of its negation is worth a million dollar.

The dual problem of deciding validity of a PC formula is *coNP-complete*.

Normal forms of PC

Definition 8.1.1.

- A *literal* is an atom p or the negation of an atom $\neg p$.
- A formula is *in negation normal form* (NNF) if it contains only $\wedge, \vee$ and $\neg$ and the negation symbol $\neg$ occurs only in literals.
- A *clause* is a disjunction of literals.
- A formula is *in conjunction normal form* (CNF) if it is a conjunction of clauses.

Alternatively, a CNF formula can be viewed as a theory consisting of clauses.

Almost all algorithms for satisfiability checking of PC require input of CNF formulas. The reason is that algorithms for formulas in CNF are easier to design and implement, and linear transformations from PC to CNF exists.

Deciding validity of CNF is linear The following simple propositions lead to an efficient algorithm to decide validity of a CNF formula. Two literals p and $\neg p$ are called *complementary*.

Proposition 8.1.1. *A clause is valid iff it contains two complementary literals.*

Proof. If a clause has no complementary literals, it is false in the structure making each disjunct false. If a clause has complementary literals p and $\neg p$, then in every structure, one of the complementary disjuncts is true. ■

Proposition 8.1.2. *A conjunction is valid iff each of its conjuncts is valid.*

Proof. If $\varphi_1 \wedge \cdots \wedge \varphi_n$ is true in a structure, then every conjunct is true in that structure. Hence, if $\varphi_1 \wedge \cdots \wedge \varphi_n$ is true in every structure, each conjunct is true in every structure. Vice versa, if every conjunct is true in a structure, then the conjunction is true in the structure. Hence if every conjunct is valid, the conjunction is valid. ■

It follows from these two propositions that a CNF formula is valid iff each of its clauses contains complementary literals.

Proposition 8.1.3. *The problem of deciding validity of a CNF formula has linear complexity in the size of the formula.*

Proof. The algorithm is to run linearly over φ and verify for each clause of φ whether it contains complementary literals. ■

The proposition suggests an algorithm to check validity of a formula φ: transform it into an equivalent CNF-formula and apply the linear algorithm. Since there is no free lunch in this universe, we can expect that the computation of a formula ψ in CNF which is equivalent to a given formula φ, is a costly worst-case operation. Indeed, if there would be an polynomial equivalence preserving transformation to CNF, then P=co-NP.

The conversion algorithm $ToCNF$ The algorithm consists of four successive rewrite phases; in each phase, one or more well-known equivalences are applied as left to right rewrite rules until no rule applies anymore:

1. $(\psi \Leftrightarrow \phi) \Leftrightarrow (\psi \Rightarrow \phi) \wedge (\phi \Rightarrow \psi)$ (*1)
2. $(\psi \Rightarrow \phi) \Leftrightarrow (\neg\psi \vee \phi)$
3. $\neg\neg\psi \Leftrightarrow \psi$
   $\neg(\psi \wedge \phi) \Leftrightarrow (\neg\psi \vee \neg\phi)$ (laws of De Morgan)
   $\neg(\psi \vee \phi) \Leftrightarrow (\neg\psi \wedge \neg\phi)$
   After this step, we obtain formulas in NNF.
4. $(\psi \wedge \phi) \vee \delta \Leftrightarrow (\psi \vee \delta) \wedge (\phi \vee \delta)$ (*2)
   distributing disjunction over conjunction

We denote the result of applying this algorithm on φ as $ToCNF(\varphi)$.

The algorithm terminates. Its output is logically equivalent with the input. Rewrite operations (*1) and (*4) create two copies of subformulas. Hence, they may double the size of the formula. Therefore, the full rewrite process blows up the formula exponentially in the worst case.

Exercise 8.1.3. *The previous statement is an argument that this algroithm is exponential but not a proof. Give a proof: find a class of formulas for which the exponential blow up occurs.*

Worst case exponential complexity does not necessarily mean that an algorithm is useless. Indeed, it depends on how often bad cases occur in practice. E.g., the standard linear programming

algorithm has exponential worst-case behaviour but until not so long ago it was unbeatable by polynomial algorithms in practical problems. Alas, in practice the above algorithm $ToCNF$ indeed behaves exponentially making it quite useless.

Remark 8.1.1. There exists various alternative normal forms for which linear algorithms for satisfiability and validity and other forms of inference exist. One is called *Binary Decision Diagrams*. They are binary tree-like formulas with syntax defined in Bachus Naur Form (BNF):

$$\psi ::= \bot|\top|(\text{ if } p \text{ then } \psi \text{ else } \psi)$$

where p is a propositional symbol, and such that all symbols in a branch of the tree differ. Such a formula is satisfiable iff it has a leaf $\top$ (true) and valid iff all leaves are $\top$. This shows that linear algorithms for satisfiability and validity exist for this format. Transforming a PC formula in a logically equivalent in this BDD format is also exponential in the worst case, but algorithms exist with much better average behaviour. The field investigating such methods is called *Knowledge Compilation*.

The Tseitin transformation We now introduce the *Tseitin transformation*, a linear algorithm for transforming a PC formula φ to CNF. It implements a simple idea, namely to replace subformulas ψ by new symbols p_ψ and express the logical connection between formulas ψ and its components by an equivalence $p_\psi \Leftrightarrow \ldots$. In a final step, all equivalences are transformed to CNF by applying the procedure $ToCNF$.

Formally, we introduce for each non-literal formula ψ a new propositional symbol p_ψ. For each non-literal formula ψ, define ψ' to be the formula obtained from ψ by replacing its component formulas α by p_α. E.g., $(\neg\alpha)'$ is $\neg p_\alpha$, $(\alpha \wedge \beta)'$ is $(p_\alpha \wedge p_\beta)$, etc. Finally define $Tseitin(\varphi)$ as the clausal theory:

$$\{p_\varphi, ToCNF(p_\psi \Leftrightarrow \psi') \mid \psi \text{ is } \varphi \text{ or a non-literal subformula of } \varphi\}$$

Proposition 8.1.4. *The Tseitin transformation has linear complexity. It preserves satisfiability. Every models of φ can be extended in a unique way to a model of $Tseitin(\varphi)$ and vice versa, each model of $Tseitin(\varphi)$ is a model of φ.*

Proof. (sketchy) The Tseitin transformation can be implemented by traversing the parse tree of input φ in one linear pass. While $ToCNF$ is exponential in general, here it is applied only to formulas with at most two connectives. Hence, the size of $ToCNF(p_\psi \Leftrightarrow \psi')$ is bounded. The size of the transformation is linear in the size of φ.

The correctness of the algorithm is based on the following observation. Let φ be a formula with strict subformula ψ. Let p_ψ be a symbol not occurring in φ. Let φ' be the formula obtained from φ by substituting the occurrence ψ by p_ψ. Then each model $\mathfrak{A}$ of φ can be expanded in a unique way with an interpretation of p_ψ to a model of $\varphi' \wedge (p_\psi \Leftrightarrow \psi)$. Indeed, the unique extension is $\mathfrak{A}[p_\psi : \psi^{\mathfrak{A}}]$. Vice versa, it is easy to see that each model of $\varphi' \wedge (p_\psi \Leftrightarrow \psi)$ is a model of φ.

Clearly the Tseitin transformation can be implemented by iterated application of the above transformation rule, followed by equivalence preserving applications of $ToCNF$. Hence, it follows that each model of φ has a unique expansion to a model of $Tseitin(\varphi)$ and vice versa, a model of the transformation is a model of the original formula. ■

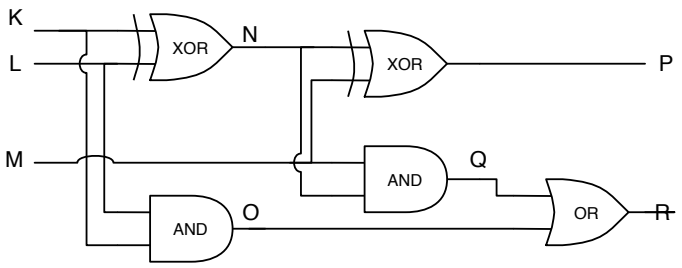
While the semantical correspondence of φ and $Tseitin(\varphi)$ is strong, the transformation does not preserve validity. This is because it introduces symbols that are not constrained by φ but are constrained by $Tseitin(\varphi)$. In particular, the basic rewrite step introduces a conjunct $p_\psi \Leftrightarrow \psi$ which is not valid. For example, take any structure interpreting ψ and expand it as $\mathfrak{A}[p_\psi : (\neg\psi)^{\mathfrak{A}}]$. In this structure, the equivalence is not satisfied.

It follows that the Tseitin transformation cannot be used in proving validity of φ. However, it can be used for satisfiability checking and model generation inference on φ.

Example 8.1.3. Consider the formula $\neg((P \Leftrightarrow Q) \vee (Q \wedge R))$. We use symbols A, B, C, D to transform for its subformulas. Its Tseitin transformation is the set of clauses in the second row.

A	$A \Leftrightarrow \neg B$	$B \Leftrightarrow C \vee D$	$C \Leftrightarrow (P \Leftrightarrow Q)$	$D \Leftrightarrow Q \wedge R$
A	$A \vee B$ $\neg A \vee \neg B$	$\neg B \vee C \vee D$ $B \vee \neg C$ $B \vee \neg D$	$\neg C \vee P \vee \neg Q$ $\neg C \vee \neg P \vee Q$ $C \vee \neg P \vee \neg Q$ $C \vee P \vee Q$	$\neg D \vee Q$ $\neg D \vee R$ $D \vee \neg Q \vee \neg R$

Exercise 8.1.4. *Compute the Tseitin transformation of the boolean circuit example.*



Summary The Tseitin transformation is widely used. Many refinements exist. E.g., avoiding multiple Tseitinisations of different occurrences of the same formula. Or to recognize certain specific subformulas (called circuits) and providing more optimal transformation.

In the end, the CNF format is not used for the efficiency of validity inference. For the simple reason that in practice, CNF formulas are never valid. The CNF format is used for satisfiability checking and model generation inference. This inference problem is still NP-complete for CNF, hence there is no computational gain. So why is CNF used? Because of its simplicity which is of great help to develop efficient algorithms.

8.2 SAT algorithms

The CNF-SAT problem The CNF-SAT problem is the satisfiability checking inference problem applied to CNF formulas. This is an NP-complete problem.

The field of SAT is a lively area, with important practical industrial applications, e.g., in verification of computer hardware, in constraint solving. Enormous progress has been made especially since 2000. In the past two decennia, the range of solvable problems went up from a few hundred variables to millions of variables and clauses. Now for some years, progress is slower but still steady.

```
Procedure UP(𝓘) {
    while there exists a unit clause with unit literal L {
        𝔄(L) := t
        if  there exists a conflict clause
        then return "Conflict"
        }
    return
    }
```

Figure 8.1: the UP procedure

A major step in this evolution has been the introduction of the CDCL algorithm which below we introduce in several steps.

8.2.1 DPLL: a basic SAT solver

The DPLL/Davis-Putnam-Logemann-Loveland algorithm [1962] is a backtracking algorithm for deciding satisfiability of a CNF formula. Although it is not efficient, it is still the basis for the current SAT solvers.

We introduce some basic concepts.

Definition 8.2.1. A partial structure $\mathcal{I}$ for (propositional) vocabulary Σ is a function from Σ to $\{\mathbf{t}, \mathbf{f}, \mathbf{u}\}$. We call $\mathbf{u}$ "undefined".

A clause C is a *conflict clause* with respect to $\mathcal{I}$ if every literal in C is false.

A clause C is a *unit clause* w.r.t. $\mathcal{I}$ if every literal is false except one literal L that is undefined. This literal is called the *unit literal*.

Partial structures arise in SAT algorithms in intermediate states, when some propositional symbols were assigned a value and others not yet.

When during a run of a SAT algorithm on a CNF formula φ, a partial structure $\mathfrak{A}$ is constructed such that some of the clauses C of φ is a conflict clause, the current $\mathfrak{A}$ cannot be expanded to a model of φ. In this case, backtracking will occur.

Likewise, if some clause is a unit clause with unit literal L, then in all models expanding $\mathfrak{A}$, L will be true. In this case, DPLL will *propagate* L. That is, it will expand $\mathfrak{A}$ so that L is true.

Unit propagation A unit propagation in the context of a partial structure $\mathfrak{A}$ is the operation of expanding $\mathfrak{A}$ to make a unit literal true. A unit propagation may cause other clauses to become unit clause and hence, may lead to a cascade of unit propagations. This propagation process is implemented by the procedure UP ("Unit Propagation") in Figure 8.1. If a run of UP assigns values to n variables, this means that $2^n - 1$ possible assignments to these variables are cut from the search space.

```
Procedure DPLL(φ) {
    I(P) := u, for all P in φ
    while true do {
        UP
        if "conflict"
        then  Backtrack
        elseif  I is a model
        then return I
        else  Decide
        }
    }
```

Figure 8.2: the DPLL procedure

The DPLL algorithm A high level description of DPLL is given in Figure 8.2. The procedure uses the following subroutines:

- **Decide**: choose a literal L that is undefined in Π and set it to true: $\mathcal{I}(L) := \mathbf{t}$. L is called a *decision literal* or *choice literal*.
- **Backtrack**: if there is no decision literal, return "Unsat". Otherwise, return to the most recent decision literal L that is true in the current partial structure and set it to false: $\mathcal{I}(L) := \mathbf{f}$.

Example 8.2.1. An example execution of DPLL.

CNF:	
	Choice: P .
	| – Propagate: $\neg R$.
	| – Propagate: $\neg Q$. Model
$\neg P \vee \neg R.$	Backtrack: $\neg P$.
$\neg Q \vee R.$	| – Choice: Q .
$P \vee \neg Q \vee \neg R.$	| | – Propagate: R .
$P \vee Q \vee R.$	Conflict: $P \vee \neg Q \vee \neg R$!
	| Backtrack: $\neg Q$.
	| | – Propagate: R . Model
	Backtrack till end.

Two models are computed: $\{P, \neg Q, \neg R\}$, $\{\neg P, \neg Q, R\}$.

Analysis of DPLL At any point of time, the state of DPLL can be represented as a sequence of labeled literals

$$\langle {L_0}^{\kappa_0} \; {L_1}^{\kappa_1} \; \ldots \; {L_n}^{\kappa_n} \rangle$$

which represents the order in which the literals L_i were made true. Each literal L_i is derived by decision or by unit propagation. The label κ_i of L_i is either "**Decision**" or the unit clause C that derived L_i.

The *time* of literal L_i in this state is i. The *level* of literal L_i in this state is the number of decision literals in the sequence up to i. All unit literals have the same level as the decision literal from which they were derived. Unassigned literals do not have a time or a level.

Notice that DPLL starts with a call to UP. If this results in conflict, Unsat is returned. Literal derived during this initial phase are literals of level 0. The first decision literal is at level 1, and so on.

Example 8.2.2. Times and levels.

$$P \vee Q \vee R$$
$$S \vee T \vee Q$$
$$\neg Q \vee R$$
$$P \vee \neg Q \vee \neg R$$
$$\neg S \vee R$$

Choice: $\neg P$.
| – Choice: $\neg S$.
| | – Choice $\neg T$.
| | | – Propagate: Q .
| | | – Propagate: R .
Conflict: $P \vee \neg Q \vee \neg R$!

	Val	$Time$	Lev
P	0	0	1
S	0	1	2
T	0	2	3
Q	1	3	3
R	1	4	3

The following observations are relevant to understand the improved algorithm that will be introduced below.

When during DPLL at level > 0 a conflict clause arises, it contains at least two (false) literals of the last level. Otherwise, the clause would have been a conflict clause or a unit clause at a previous level. In the example, this is $\neg Q$ and $\neg R$.

When a unit literal L was derived at the current level > 0, it was by a unit clause with at least 2 literals of the same level: namely L which is true and at least one literal that is false. Otherwise, this clause would have been a unit clause at a previous level. Notice that clauses of length 1 are unit clauses of level 0.

A disadvantage of DPLL is that it does not learn; it may make choices for a group of variables that lead to failure, and after backtracking to older levels, it may keep repeating the same bad choices for these variables over and over again.

8.2.2 Conflict-Driven Clause learning (CDCL)

This is an optimization of DPLL that performs *resolution* in a smart way to cut exponential parts of the search tree.

Definition 8.2.2. The resolution of two clauses $\psi = p \vee L_1 \vee \cdots \vee L_n$ and $\phi = \neg p \vee L'_1 \vee \cdots \vee L'_m$ on p is the *resolvent* $\delta = L_1 \vee \cdots \vee L_n \vee L'_1 \vee \cdots \vee L'_m$.

Resolution is sound: $\psi \wedge \phi$ logically entails δ.

Adding resolvents of clauses of a CNF theory to the theory preserves equivalence and has no impact on the set of computed models. However, it may improve considerably the efficiency of the DPLL algorithm. Hence, it seems like a good idea to add resolvents to the theory. The question is : which resolvents? We cannot add all resolvents: there are too many of them. The number of resolvents of a CNF theory is exponential in the number of clauses.

In the *Conflict-driven clause learning* algorithm (CDCL), resolvents are constructed that directly aid the solving process. They cut away potentially large parts of the search space by allowing deep backtracking. When a conflict clause arises, resolution is applied to construct a clause that gives the "reason" for the conflict. This clause is called the *learned clause*, and allows to perform *backjumping*, i.e. to backtrack over multiple levels. This may cut away exponential parts of the search tree. Moreover, the clause is stored to avoid that a failed assignment is repeated.

The CDCL procedure is activated when a conflict arises. Below its operation is described.

First resolution step Assume that during execution, a conflict arises due to the conflict clause $L_1 \vee \cdots \vee L_n$. Recall that all literals have an "age", which is their time stamp. Assume that the clause is ordered from young to old, in which case L_1 is the youngest literal. Recall that the conflict clause contains at least two literals of the current level, which implies that L_1 and L_2 are of the current level. L_1 is not the negation of the decision literal since that literal is the oldest literal of the current level and L_1 is younger than L_2. Hence, L_1 is a unit literal.

L_1 is the negation of a unit literal $\neg L_1$ of the current level that was derived by a unit clause $\neg L_1 \vee L'_2 \vee \cdots \vee L'_m$. CDCL performs resolution on the conflict clause and the unit clause on L_1 and removes doubles. The result is a new clause which we denote as L_1" $\vee \cdots \vee L_k$". Again, we assume that literals are ordered from young to old. Below, this operation is called: **ResolveYoungest**.

Since $\neg L_1$ was the only true literal in both clauses and was resolved away, this resolution step produces again a conflict clause.

Notice, since the conflict clause contained at least two false literals of the current level, the returned conflict clause still contains at least one false literal of the current level. In particular, the youngest literal L_1" is of the current level.

Example 8.2.3. Example 8.2.2 continued. It constructs the structure with P, S, T false and Q, R true.

$$P \vee Q \vee R.$$
$$S \vee T \vee Q.$$
$$\neg Q \vee R.$$
$$P \vee \neg Q \vee \neg R.$$
$$\neg S \vee R.$$

Choice: $\neg P$.
| – Choice: $\neg S$.
| | – Choice $\neg T$.
| | | – Propagate: Q .
| | | – Propagate: R .
Conflict: $P \vee \neg Q \vee \neg R$!

ResolveYoungest($P \vee \neg Q \vee \neg R$):
- Reorder to $\neg R \vee \neg Q \vee P$
- Resolve with unit clause of R: $\neg Q \vee R$.
- Remove double $\neg Q$.
- Return $\neg Q \vee P$.

The returned clause is a conflict clause.

ResolveYoungest takes as input a conflict clause and returns a new conflict clause with its youngest literal L_1" a false literal of the current level.

In general, *ResolveYoungest* can be applied to an arbitrary conflict clause $L_1 \vee \ldots$ with a literal L_1 of the current level, provided this literal is the negation of a unit literal. In that case, $\neg L_1$

```
Procedure ClauseLearning(ConflictClause) {
    while ConflictClause contains ≥ 2 literals of current level {
        ConflictClause = ResolveYoungest(ConflictClause)
        }
    return ConflictClause
    }
```

Figure 8.3: the ClauseLearning procedure

was derived by a unit clause $\neg L_1 \vee K_2 \vee \ldots$. Again, resolution is possible. As we observed before, the unit clause $\neg L_1 \vee K_2 \vee \ldots$ contains at least two literals of the current level. It follows that the resolvent contains at least one false literal of the current level: K_2 and there may be more of them. Hence, an invariant of iterated application of *ResolveYoungest* is that the produced clauses are conflict clauses with the youngest literal of the current level.

This process may come to an end when a conflict clause is produced in which the only and youngest literal is the negation of the decision literal of the current level. The decision literal of the current level has no unit clause and cannot be resolved away.

Thus, *ResolveYoungest* can be iterated until the only literal of the current level is the negation of the decision level. However, CDCL does something smarter. It iterates *ResolveYoungest* until a conflict clause is obtained with exactly one literal of the current level. This clause is called the *learned clause* or the *Unique Implication Point* (UIP). The procedure to compute it is called **ClauseLearning** and is represented in Figure 8.3. Below, we illustrate this strategy and discuss its merits.

Example 8.2.4. Example 8.2.2 continued.

$$
\begin{array}{c}
P \vee Q \vee R. \\
S \vee T \vee Q. \\
\neg Q \vee R. \\
P \vee \neg Q \vee \neg R. \\
\neg S \vee R.
\end{array}
\qquad
\begin{array}{l}
\text{Choice: } \neg P \ . \\
\mid \text{– Choice: } \neg S \ . \\
\mid \quad \mid \text{– Choice } \neg T. \\
\mid \quad \mid \quad \mid \text{– Propagate: } Q \ . \\
\mid \quad \mid \quad \mid \text{– Propagate: } R \ . \\
\text{Conflict!}
\end{array}
$$

ClauseLearning($P \vee \neg Q \vee \neg R$):

- Apply *ResolveYoungest*: resolve $\neg R$ with $\neg Q \vee R$.
- We obtain $\neg Q \vee P$. This is the UIP. Note that $\neg Q$ is not the decision literal of the current level.
- **Return this clause.**

Here one resolution step suffices to produce the UIP $P \vee \neg Q$. Its youngest literal is $\neg Q$. It is the only one of the current level. The second literal in the UIP is P which is the negation of the oldest decision literal.

All clauses computed during the process are entailed by the theory and are conflict clauses in the current assignment (all literals are false).

At each step, we eliminate the youngest literal of the formula and replace it by older ones, hence the youngest literal in the conflict clause becomes strictly older. This cannot go on forever, hence this algorithm terminates.

Procedure Backjump(LearnedClause) {
Determine m from LearnedClause.
Undo all assignments to literals of level $m+1, m+2, \ldots, n$.
}

Figure 8.4: the Backjump procedure

At worst, the algorithm runs until the only literal of the current level is the negation of the current choice literal. But it may stop earlier as illustrated in the example.

The output of clause learning is a clause $L_1 \vee \cdots \vee L_k$ with one literal L_1 of the current level and all other literals of older levels. This is the *learned clause* (the UIP).

What to do with this clause? There are two uses. First, it is added to the clause database, to avoid that later executions repeat the erroneous derivation. Second, it is used for backjumping, as explained next.

Backjumping: non-chronological backtracking Suppose the learned clause is (ordered in age from young to old):

$$L_1 \vee \mathbf{L_2} \vee \cdots \vee L_k (k \geq 1)$$

L_1 is the youngest, of the current level n; L_2 is the second youngest, of level $m < n$. At the current level n, this is a conflict clause.

CDCL performs a special sort of backtracking to the level m of $\mathbf{L_2}$. Backtracking to level m ($0 \leq m < n$) means: undoing all assignments to variables of level $m+1, m+2, \ldots, n$. In contrast to normal backtracking, all assignments at level m are kept.

Since all literals of the UIP except one are of level m or older, m is the oldest level where the learned clause is a unit clause with unit literal L_1. After undoing all assignments of levels $m+1, m+2, \ldots, n$, backtracking proceeds by restarting the unit propagation process at level m with the learned clause as new unit clause. After the call to this procedure, UP is called again, as in DPLL. The backjump procedure used by CDCL is given in Figure 8.4.

Example 8.2.5. Continuation of the example

	Choice: $\neg P$.
	| – Choice: $\neg S$.
	| | – Choice $\neg T$.
$P \vee Q \vee R$.	| | | – Propagate: Q .
$S \vee T \vee Q$.	| | | – Propagate: R .
$\neg Q \vee R$.	Conflict! Learn $\mathbf{P \vee \neg Q}$
$P \vee \neg Q \vee \neg R$.	Backtrack over $\neg T$ and $\neg S$.
$\neg S \vee R$.	| – Propagate $\neg Q$.
$\mathbf{P \vee \neg Q}$	| – Propagate: R .
	| | – Choice $\neg S$.
	| | | – Propagate T . Model!

Exercise 8.2.1. *Continue the execution.*

This is a *backjumping* algorithm, backtracking till level $m+1$. Moreover, rather than switching the choice literal of the backtrack level as in chronological backtracking, CDCL continues with

unit propagation at level m, with the guarantee that there is at least one new unit clause, namely the UIP.

Why is it so good to stop at the first UIP? After all, we could continue the iteration of **ResolveYoungest**. Each conflict clause with one literal of the current level computed by iterating **ResolveYoungest** is a candidate clause for backjumping. If we iterate all the way, we obtain a conflict clause with only one literal of the current level, namely the decision literal. Also this clause can be used for backjumping.

It is easy to see that it is best to stop with the UIP. Indeed, during iteration of **ResolveYoungest**, literals of lower level are not resolved away. Hence, the set of them grows with each resolution step. This has two disadvantages. First, longer clauses do not propagate as often as short clauses. Second and probably worse, after the UIP was obtained, later resolution steps might introduce literals of a more recent level m' than the backtrack level m of the UIP. In this case, backjumping is less deep than with the UIP, to level m' rather than to level m.

This is illustrated below.

Example 8.2.6. ResolveYoungest past the UIP.

$P \vee Q \vee R.$	Choice: $\neg P$.
$S \vee T \vee Q.$	| – Choice: $\neg S$.
$\neg Q \vee R.$	| | – Choice $\neg T$.
$P \vee \neg Q \vee \neg R.$	| | | – Propagate: Q .
$\neg S \vee R.$	| | | – Propagate: R .
	Conflict!

ClauseLearning-v2($P \vee \neg Q \vee \neg R$):
- Resolve on $\neg R$ with $\neg Q \vee R$.
- Obtain $P \vee \neg Q$ (the UIP).
- Resolve on $\neg Q$ with $S \vee T \vee Q$
- Obtain $P \vee S \vee \mathbf{T}$: contains only the choice literal $\mathbf{T}$ of current level.
- **Return this clause.**

The extra resolution step has introduced additional literal S of a non-current level, but S is of younger level than P. If $P \vee S \vee T$ is used for backjumping rather than $P \vee \neg Q$, backjumping will be to the level of S which is less deep.

Deeper backjumping may remove an exponentially larger part of the search tree.

Example 8.2.7. Backjumping with the alternative learned clause.

	Choice: $\neg P$.
	| – Choice: $\neg S$.
$P \vee Q \vee R.$	| | – Choice $\neg T$.
$S \vee T \vee Q.$	| | | – Propagate: Q .
$\neg Q \vee R.$	| | | – Propagate: R .
	Conflict! Let's add $\mathbf{P} \vee \mathbf{S} \vee \mathbf{T}$.
$P \vee \neg Q \vee \neg R.$	Backtrack over $\neg T$.
$\neg S \vee R.$	| | – Propagate T.
$\mathbf{P} \vee \mathbf{S} \vee \mathbf{T}$	| | – Propagate: Q .
	...

This is less efficient.

```
Procedure CDCL(φ) { % prints all models; UNSAT if list is empty
    Initialize I(P) ← u, for all P ∈ Σ
    while true {
        UP
        if I is a model
        then { Print I
               ConflictClause := NoModel(𝔄)
               Conflict := true
               Add ConflictClause to Clause database
               }
        if conflict and current level is 0
        then { return "Unsat" }
        if conflict and current level is > 0
        then { LearnedClause := ClauseLearning(ConflictClause)
               Backjump(LearnedClause)
               }
        else { % no conflict and there are undefined symbols
               Choose an undefined literal L and assign I(L) := t
               }
        }
}
```

Figure 8.5: CDCL procedure

When does CDCL stop? CDCL stops after finding a model or when it discovers unsatisfiability. The latter occurs when it backtracks to level $m = 0$ and subsequent UP computes a conflict clause. In that case, CDCL returns $Unsat$, just like DPLL.

Computing multiple models To adapt CDCL to generate multiple models, the standard technique is as follows. When a model $\mathfrak{A}$ is found, a clause is derived that forbids this model.

One such a clause is $\vee_{L \in \mathfrak{A}} \neg L$, the disjunction of negations of true literals of $\mathfrak{A}$. A shorter and hence better clause is to include only the negation of the decision literals of $\mathfrak{A}$. We denote this clause as **NoModel($\mathfrak{A}$)**. The full CDCL algorithm for computing all models is in Figure 8.5

Implementation of SAT-solvers The fastest SAT solvers are optimised to the extreme:

- arrays for constant time access;
- datastructures were developed such that important parts fit into the *cache* of the processor as much as possible, to avoid swapping $\Rightarrow\times$ 10-40 faster;
- no automated memory management, no recursive procedures; the program is in control of all memory-management;
- no algorithms more than linear time, except for the basic backtracking algorithm (which is exponential in the worst case);
- implemented in C or C++, no Java.

A number of crucial optimisation techniques are discussed now.

Unit propagation: 2 Watched Literals How to detect conflict and unit clauses without rechecking each clause after each assignment? The 2 watched literal technique serves to give access to clauses only when really necessary.

The technique "watches" 2 non-false literals (true or unknown) in each clause. E.g.,

$$\underline{P^{\mathbf{u}}} \vee \underline{Q^{\mathbf{u}}} \vee \neg R^{\mathbf{u}} \vee S^{\mathbf{u}}$$

Note: atoms are annotated here with their truth value.

When a watched literal is made **false**, action is required. First an undefined non-watched literal is searched. If found, it is selected as a new watch. E.g., Q is made false, $\neg R$ is the new watch:

$$\underline{P^{\mathbf{u}}} \vee \underline{\mathbf{Q^{f}}} \vee \neg R^{\mathbf{u}} \vee S^{\mathbf{u}} \quad \longrightarrow \quad \underline{P^{\mathbf{u}}} \vee Q^{\mathbf{f}} \vee \underline{\neg R^{\mathbf{u}}} \vee S^{\mathbf{u}}$$

If no candidate watch is found, we detected that the clause is a unit clause or conflict. If the other watched literal is unknown, it is a new unit literal. E.g., in a state where all literals except the watch P are false:

$$\underline{P^{\mathbf{u}}} \vee \underline{\mathbf{Q^{f}}} \vee \neg R^{\mathbf{t}} \vee S^{\mathbf{f}} \quad \longrightarrow \quad \text{unit propagation } P$$

To implement, literals have constant time access to the clauses where their negation is watched. Hence, when the literal is made true, it has direct access to all clauses where potential conflict or unit propagation takes place.

UP Heuristics There are several heuristic strategies to implement UP:
- Breadth first: use oldest unit clauses to propagate first.
- Depth first: use youngest unit clauses first

Both strategies are equivalent in the sense that a call to UP has the same effects. If one leads to a conflict, so will the other. However, it may be a different conflict clause. If none leads to conflict, they derive exactly the same unit literals.

Example 8.2.8.

$\neg A \vee B$
$\neg A \vee C$
$\neg B \vee D$
$\neg B \vee \neg D$
$\neg B \vee \neg C$

- Choose A
- Breadth first UP $\Rightarrow$ conflict: $\neg B \vee \neg C$
- Depth first UP $\Rightarrow$ conflict: $\neg B \vee \neg D$

It may therefore seem that the chosen strategy is of little importance. Unexpectedly, in CDCL it turns out to have a big impact. The reason is that the conflict clause produced by breadth first tends to contain "older" literals. When combined with conflict clause learning, this causes *deeper* backtracking. Backtracking after depth first UP is shallower.

Decision Heuristics: VSIDS VSIDS is a heuristic fully named *Variable State Independent Decaying Sum*. Each literal has a counter initialized to 0. When a conflict clause is added, the counters of its literals are incremented. At a decision point, the unassigned variable with the highest counter is chosen. Periodically, after N choices, all the counters are divided by a constant.

VSIDS is called a conflict-driven decision strategy. The effect of this strategy is that variables appearing in recent conflict clauses get higher priority. As a consequence, search tends to focus on difficult parts of the search space, and this proved to be a major advantage. This strategy dramatically improved performance by an order of magnitude.

The implementation is by maintaining a list of literals. Updating is only needed when adding a conflict clause.

Clause deletions After a while, memory gets full with learned clauses. At regular occasions, clauses that were not recently used for unit propagation are deleted.

Random restarts The order of the selection of variables is important. A wrong initial selection can lead to exponentially larger search trees. Heuristics do not always make correct choices. However, due to the clause learning, more and more information becomes available; heuristic information about what are the important variables is accumulated in the VSIDS counters of literals.

During a run of CDCL, this information is not fully exploited because backtracking never restarts the search from scratch. Therefore, in modern solvers, search is stopped and is restarted using the learned clauses at regular times. The restarts happen at an exponential rate. That is, the n'th run gets $O(2^n)$ time. Hence, initially the number of restarts is high, it slows down quickly.

8.2.3 Summary

CDCL made SAT solvers applicable to theories that are orders of magnitude larger than was feasible before.

Currently, progress has slowed down but is still steady. New techniques emerge. E.g., symmetry breaking turns out to be powerful tool. The symmetry breaker BreakID developed at KRR combined with state of the art SAT solvers won several times in the SAT competition. New techniques combine CDCL with parallel execution by splitting up the search space in a large number of small search problems. A solver of this kind build a two hundred terabyte math proof.

`https://www.nature.com/news/two-hundred-terabyte-maths-proof-is-largest-ever-1.19990`

8.3 Other forms of inference in PC

Most research on inference in PC is concentrated on the satisfiability/model generation problem. This problem is called the *SAT problem*.

Other useful forms of inference are:

- Deductive problems: deciding validity and entailment. They can be reduced to SAT problems.
- MAXSAT

  MAXSAT:
  - Input: a PC theory T
  - Output: a $\subseteq$-maximal PC theory $T' \subseteq T$ that is satisfiable.

  This is useful when a problem is overconstrained and we want a solution where as many of the constraints as possible are satisfied.

- Model counting

  Model counting:
  - Input: a PC theory T
  - Output: the number of models of T.

  This form of inference is useful in the context of probabilistic reasoning. The ratio of the number of models of $T \cup \{\varphi\}$ over the number of models of T is the probability of φ in the context of the domain axiomatized by T.

8.4 Finite model expansion in IDP

IDP is a knowledge base system that supports multiple forms of inference for FO(.) theories and structures. The main form of inference is model expansion. We finish this section with a brief description of model expansion in IDP.

Model expansion is the following form of inference.

Model expansion:
- Input: theory T, partial structure $\mathcal{I}$;
- Output: an expansion $\mathfrak{A}$ of $\mathcal{I}$ that is a model of T, or "Unsat" if there is no expansion.

The input for the model expansion of IDP is a theory T and a *partial structure* $\mathcal{I}$ of the vocabulary Σ of T. The approach for model expansion in IDP is by *ground and solve*. IDP reduces T together with a $\mathcal{I}$ to a propositional theory $T^g_{\mathcal{I}}$ called the *grounding of* T *in* $\mathcal{I}$, and then applies (extended) SAT solving to $T^g_{\mathcal{I}}$.

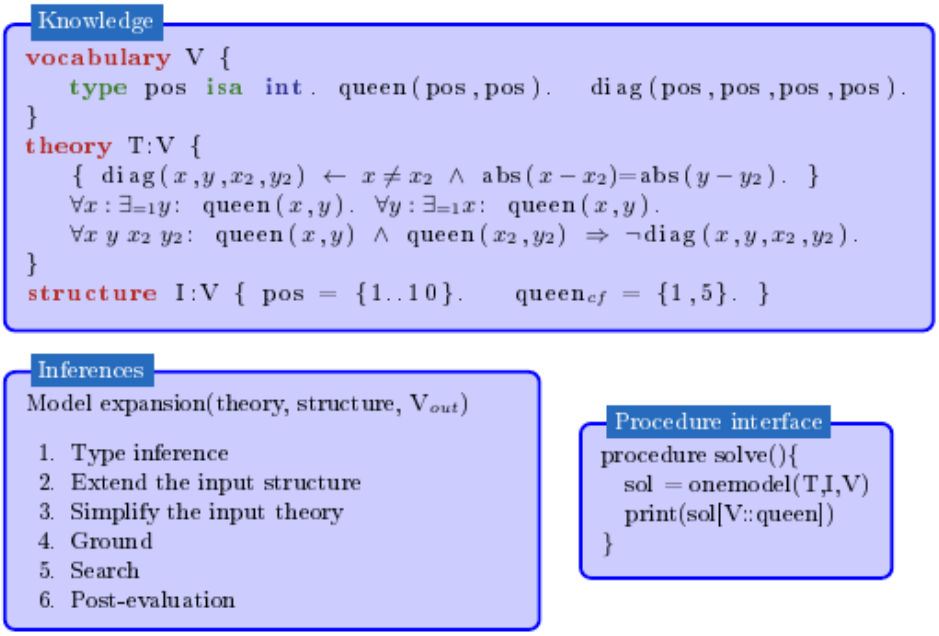


Fig. 1. Running example: NQueens with n = 10, where no queen (cf stands for "certainly false") is placed at $(1,5)$.

Figure 1 shows an application of IDP to the n-queens problem. The box *Knowledge* contains vocabulary, theory and structure. The box *Procedure interface* specifies a call to the model expansion procedure of IDP. The box *Inferences* specifies the six different computational steps that take place. Below we provide some detail on these steps.

(Partial) structure A structure declaration in the IDP language specifies a partial structure with a finite set as value for all types. It specifies values or *partial values* for constant, predicate and function symbols.

In Figure 1, the structure I specifies the value $\{1,\ldots,10\}$ for type *pos* and a partial value for $queen(pos,pos)$, namely that $queen(1,5)$ is false and all other atoms $queen(i,j)$ are undefined. This is expressed by the notion $queen_{cf}$ where cf stands for "certainly false". The value of $diag/4$ is unknown in I.

Model expansion for this input will compute models in which $queen(1,5)$ is false.

Type inference. IDP derives a type for every occurrence of every variable in the theory. This is done (mostly) in the standard way, by matching type declarations of symbols with the terms that occur as arguments in the theory. If the system derives a complete typing, it accepts the theory. Otherwise, it generates a type error. Type inference is useful for debugging and necessary for determining the domains of all logical variables in formulas. The use of subsorts in IDP makes type inference more complex and subtle. The manual discusses it.

Extend the input structure Prior to grounding, the IDP system applies several procedures to expand/refine the input structure. Expansion does not increase the size of the domains of

types but it turns unknown facts into known ones. The benefit of refining the input structure $\mathfrak{A}$ is that grounding leads to a propositional theory of smaller size.

One technique to expand $\mathfrak{A}$ is to compute defined predicates of a definition whose parameters are all known. In the example, $diag(pos, pos, pos, pos)$ is such a predicate. $diag(x, y, u, v)$ represents that positions (x, y) and (u, v) are on the same diagonal. This relation can be computed prior to solving. The result is stored in the structure as a value of the predicate $diag$.

Another approach is to refine $\mathfrak{A}$ by (symbolic) propagation methods. E.g., assume that it is given in T or $\mathfrak{A}$ that there is certainly a queen on position (4,6). This can be expressed by an assignment $queen_{ct} = \{(4, 6)\}$. By symbolic propagation, new facts can be derived: that all unknown atoms $queen(i, j)$ describing positions on the same row, column, and diagonals are certainly false. A symbolic propagation method is run on the theory to compute such facts and to expand the current partial interpretation $\mathfrak{A}$.

Simplify the input theory This operation serves to bring the theory in a normalized form to facilitate grounding. This includes the simplification of function terms. All nested function terms are eliminated from the theory. The resulting theory contains only atoms of the form $P(x_1, \ldots, x_n)$ and $F(x_1, \ldots, x_n) = x$ with variables, no terms. This can be achieved by iterated application of the following equivalence preserving rewrite rules:

$$\begin{array}{lcl} P(\ldots t \ldots) & \longrightarrow & \exists x(t = x \wedge P(\ldots x \ldots)) \\ t = F'(\ldots) & \longrightarrow & \exists x(t = x \wedge F'(\ldots) = x) \\ F(\ldots F'(\ldots) \ldots) = t & \longrightarrow & \exists x(F(\ldots x \ldots) = t \wedge F'(\ldots) = x) \end{array}$$

Another technique is to insert bounds for the quantified variables. IDP has a symbolic propagation module to derive bounds for quantified variables. In particular, for each quantified subformula $\forall \bar{x}\psi$ and $\exists \bar{x}\psi$, a bounds formula $\Psi_b[\bar{x}]$ is computed such that the value of $\{\bar{x} \mid \Psi_b[\bar{x}]\}$ is determined by $\mathcal{I}$. These bounds are computed in such a way that the following equivalences hold :

$$\forall \bar{x}\psi \Leftrightarrow \forall \bar{x}(\Psi_b[\bar{x}] \Rightarrow \psi)$$
$$\exists \bar{x}\psi \Leftrightarrow \exists \bar{x}(\Psi_b[\bar{x}] \wedge \psi)$$

During grounding, the value of the bound formulas are computed and used to instantiate the variables. An often essential component of these bound formulas are types.

To illustrate the principle, assume the theory contains an axiom:

$$\forall x\ y(R(x, y) \Rightarrow P(x, y))$$

where P is an interpreted symbol of $\mathcal{I}$ and R is not interpreted. The bounds module of IDP will derive that $P^{\mathcal{I}}$ is an upperbound of R. In this case, $P(x, y)$ is a bound formula for

$$\exists x\ y(R(x, y) \wedge \Psi)$$

During grounding, x, y are substituted only by values $(a, b) \in P^{\mathcal{I}}$.

Ground The grounding procedure consists of several steps. The first step is to eliminate quantifiers and variables and exploits the bounds as follows:

$$\begin{array}{lcll} \forall \bar{x} : \Psi_b[\bar{x}] \Rightarrow \varphi[x] & \longrightarrow & \bigwedge_{\bar{a} \in \{\bar{x} \mid \Psi_b[\bar{x}]\}^{\mathcal{I}}} \varphi[a] & \text{(a conjunction)} \\ \exists \bar{x} : \Psi_b[\bar{x}] \wedge \varphi[x] & \longrightarrow & \bigvee_{a \in \{\bar{x} \mid \Psi_b[\bar{x}]\}^{\mathcal{I}}} \varphi[a] & \text{(a disjunction)} \end{array}$$

The second step is to simplify the resulting formulas and definitions using the Tseitin transformation.

Another simplification is that atoms with known value in $\mathcal{I}$ are replaced by their value, and the formula is simplified. E.g., if $P(d_1)$ is true in $\mathcal{I}$, the formula $P(d_1) \vee \Psi$ is replaced with $\mathbf{t}$. In case Ψ is large, a serious benefit is made here. Such techniques are indispensable.

In the final step, atoms $P(d_1, \ldots, d_n)$ and $F(d_1, \ldots, d_n) = d$ are transformed in propositional symbols. Here $d_1, \ldots, d_n, d$ are domain elements specified in $\mathfrak{A}$. The result is a propositional theory in a normalized version of PC(.), extending CNF with ground versions of aggregate expressions, constraints and definitions.

Solve The solver of IDP is minisatID. It is called on the ground theory $T^g_{\mathfrak{A}}$ to compute one or more models of it. minisatID is an extension of a SAT solver called minisat, with additional propagators for (ground) definitions, aggregate expressions and constraint variables and constraints.

Post-evalutation Models of the ground theory are translated back to models of the input theory T.

International context IDP belongs to a family of language and system paradigms with increasinlgy expressive declarative languages:

- Answer Set Programming (an extension of logic programming).
- SAT Modulo Theories ; Microsoft built a powerful satisfiability system Z3 that is used in verification.
- Constraint Programming systems for rich Constraint Programming languages such as MiniZinc systems.
- Satisfiability checking systems for verification: ProB, TLA+.

Two aspects are unique for IDP. First, it supports inductive definitions of a very rich kind (only Answer Set Programming has something similar). Second is that IDP is explicitly conceived as a knowledge base system, a system supporting a range of inferences on the same knowledge base.

8.5 Important for exam

```
Big question: I may ask to execute the CDCL algorithm on a
specific SAT theory, and to explain the details. Knowledge of
optimisations is a plus.

Small questions:
- why the equivalence preserving translation of Proposiotional calculus is exponential
-  the Tseitin transformoration, why it is not equivalence preserving.
```

What I think you need to know/understand for solving these questions:

- Knowledge of the standard equivalence preserving transformation to CNF; why it is exponential; why it is to be expected that polynomial transformations do not exist.

- Knowledge of tseiting transformation to CNF; why it is not equivalence preserving; the weaker correctness result.

- Knowledge of DPLL and CDCL

Chapter 9

Algorithms for CTL and LTL model checking

This chapter is based on the corresponding chapter of the book of Huth and Ryan.

9.1 CTL-model Checking algorithm

In this section, we see an algorithm for solving the following inference problem:

> The CTL model checking problem:
>
> - input: a transition structure $\mathcal{M} = \langle S, \to, L \rangle$, a state $s_0 \in S$, a CTL formula ϕ;
> - output: $(\mathcal{M}, s_0 \models \phi)$ (true if it holds, false otherwise).

Instead, we resolve a related problem.

A related problem:

> - Input: $\mathcal{M}$, a CTL-formula ϕ
> - Output: $\{s \in S \mid \mathcal{M}, s \models \phi\}$. We denote this set as S_ϕ.

The output of the second problem consists of all states $s \in S$ in which the CTL state formula ϕ is satisfied. Clearly, the model checking problem can be reduced to the second sort of problem. The algorithm presented below solves both sorts of problems.

Adequate set The CTL formulas supported by the algorithm are the formulas build from the the adequate set $\{\mathrm{EG}, EU, \mathrm{EX}, \wedge, \neg, \bot\}$.

The algorithm The following proposition specifies the CTL satisfaction relation as defined in CTL*.

Proposition 9.1.1.

- $\mathcal{M}, s \models p$ *iff* $p \in L(s)$.
- $\mathcal{M}, s \models \psi_1 \wedge \psi_2$ *iff* $\mathcal{M}, s \models \psi_1$ *and* $\mathcal{M}, s \models \psi_2$.
- $\mathcal{M}, s \models \neg\psi_1$ *iff* $\mathcal{M}, s \not\models \psi_1$.
- $\mathcal{M}, s \models \mathrm{EX}\,\psi_1$ *if there exists* $s \to s_1$ *such that* $\mathcal{M}, s_1 \models \psi_1$.
- $\mathcal{M}, s \models \mathrm{E}(\psi_1\,\mathrm{U}\,\psi_2)$ *iff there exists a finite path* $s = s_0 \to s_1 \to \ldots \to s_n$ $(n \geq 0)$ *such that* $\mathcal{M}, s_i \models \psi_1$ *for every* $i \in \{0, \ldots, n-1\}$ *and* $\mathcal{M}, s_n \models \psi_2$.
- $\mathcal{M}, s \models \mathrm{EG}\,\psi_1$ *iff there exists an infinite path* $s_0 \to s_1 \to \ldots$ *such that* $s = s_0$ *and* $\mathcal{M}, s_i \models \psi_1$ *for every* $i \in \mathbb{N}$.

The proposition suggests to compute $\mathcal{M}, s \models \phi$ recursively on the structure of ϕ. The algorithm computes sets $S_\psi = \{s \in S \mid \mathcal{M}, s \models \psi\}$ recursively, for increasing subformulas ψ of ϕ.

Below, we discuss the steps in the algorithm and their complexity. Let V be the number of states and E the number of edges in $\to$. We make the following complexity assumptions about operations on datastructures:

- For each $s \in S$, the sets $\{s' \mid s' \to s\}$ and $\{s' \mid s \to s'\}$ can be generated in time linear in their size.
- For each $s \in S$ and $p \in \Sigma$, checking $p \in L(s)$ is $O(1)$ (constant time).
- For each S_ψ, checking $s \in S_\psi$ and adding s to S_ψ is $O(1)$.

It is routine to build datastructures that satisfy these conditions.

The algorithm computes S_ψ by a case analysis on ψ.

- $\psi = \bot$: $S_\psi = \emptyset$;
- $\psi = p \in \Sigma$: $S_\psi = \{s \in S \mid L(s) = p\}$;
  To compute $S_{\neg\psi_1}$, loop over the elements $s \in S$ and check $p \in L(s)$. S_p can be computed in $O(V)$ since checking $p \in L(s)$ and adding s to $S_{\neg\psi_1}$ is $O(1)$.
- $\psi = \neg\psi_1$: $S_\psi = S \setminus S_{\psi_1}$;
  Loop over the elements $s \in S$; if $s \notin S_{\psi_1}$ add s to S_ψ. This is $O(V)$.
- $\psi = \psi_1 \wedge \psi_2$: $S_\psi = S_{\psi_1} \cap S_{\psi_2}$;
  Loop over the elements $s \in S$; if $s \in S_{\psi_1}, s \in S_{\psi_2}$, add s to S_ψ. This is $O(V)$.
- $\psi = \mathrm{EX}\,\psi_1$: $S_\psi = \{s \in S \mid \exists s' : s \to s' \wedge s' \in S_{\psi_1}\}$;
  Loop over the elements $s' \in S_{\psi_1}$; for every $s \to s'$, add s to S_ψ. This is $O(V + E)$.

  Exercise 9.1.1. *In the worst case, the sets* S_{ψ_1} *and* $\{s \mid s \to s'\}$ *contain* $O(V)$ *elements. Why is the complexity not* $O(V^2)$*, i.e., quadratic? Argue.*

- $\psi = \mathrm{E}(\psi_1 \,\mathrm{U}\, \psi_2)$: $S_\psi = \{s \in S \mid \exists s_0, \ldots, s_n : s = s_0 \to s_1 \to \ldots \to s_n \wedge \forall i \in \{0, n-1\} : s_i \in S_{\psi_1} \wedge s_n \in S_{\psi_2}\}$.
  The set S_ψ is computed by the following algorithm. It uses a queue datastructure Q for which pushing and popping an element is $O(1)$.

  $S_\psi := Q := S_{\psi_2}$;
  while $Q \neq \emptyset$ {
  pop s from Q;
  for every $s' \to s$, if $s' \in S_{\psi_1}$ add s' to S_ψ and to Q.
  }
  return S_ψ

  This algorithm computes S_ψ in $O(V + E)$.

- $\psi = \mathrm{EG}\, \psi_1$: $S_\psi = \{s \mid \exists \pi = s_0 \to s_1 \to \cdots : s = s_0 \wedge \forall i \in \mathbb{N} : s_i \in S_{\psi_1}\}$.
  The set S_ψ can be computed as follows.

  $S_\psi := S_{\psi_1}$
  (1) **for** $s \in S_\psi$ {
  if s has no edge $s \to s'$ such that $s' \in S_\psi$
  then delete s from S_ψ; go to (1)
  }
  return S_ψ

  Every time an element of S_ϕ is deleted, the for loop is restarted. The for loop takes $O(V + E)$ in the worst case. The for loop is restarted potentially V times. Hence, this algorithm runs in $O(V \times (V + E))$.

Theorem 9.1.1. *For each CTL formula ϕ, $s \in S_\phi$ if and only if $\mathcal{M}, s \models \phi$*

No proof. This theorem is proven by induction on the structure of ψ, using the definition of satisfaction of path and state formulas in CTL* (Chapter 4).

Remark 9.1.1. In the book of Huth and Ryan, a different way is used to represent this algorithm. Rather than computing sets S_ϕ, Huth and Ryan extend the set of labels of a state s with CTL formulas that are true in s. Here S_ϕ corresponds to the set of states labeled with ϕ. Of course, this boils down to the same.

Complexity of the algorithm Assume that ψ has f subformulas. A run is needed over all f subformulas of ψ. Each step is at most $O(V \times (V + E))$. It follows that the algorithm has complexity $O(f \times V \times (V + E))$.

This algorithm is quadratic in the size of $\mathcal{M}$. This is too expensive. Indeed, the size of $\mathcal{M}$ tends to be very large.

A more efficient computation of $S_{\mathrm{EG}\,\psi_1}$ The only non-linear step in the algorithm is for $\mathrm{EG}\,\psi_1$. Here is an alternative algorithm to compute this set.

- Compute $\mathcal{M}'$ by deleting all states in which ψ_1 is false;

$$S' = S_{\psi_1} \text{ and } \rightarrow' = \{(s, s') \in S'^2 \mid s \rightarrow s'\}$$

- Compute all maximal strongly connected components (SCC's) of $\rightarrow'$.
  A strongly connected component of a graph is a set of states such that each state in it is reachable from each other state. Strongly connected components of a graph can be computed with Tarjans algorithm in $O(V + E)$.
- Compute $S_{\mathrm{EG}\,\psi_1}$ as the set of states in $\mathcal{M}'$ that can reach a strongly connected component.

This algorithm computes in $O(V + E)$.

Exercise 9.1.2. *Explain why this algorithm is correct.*

In conclusion, the complexity of the refined algorithm is $O(f \times (V + E))$ and is linear in the size of $\mathcal{M}$.

The state explosion problem The algorithm is linear, but unfortunately the model $\mathcal{M}$ is typically exponential in the number of variables and the number of processes (i.e., components that execute in parallel). Adding one boolean variable doubles the size of the model and the cost to verify it. A lot of research is going on to overcome this problem.

9.1.1 CTL model checking with fairness

A fairness constraint ψ is a path constraint that ψ should be true infinitely often. As discussed before, such constraints cannot be expressed in a state transition graph.

Some model checking systems support adding explicit fairness constraints to the specification. One example is the system NuSMV. In that case, the specification of the dynamic system is a pair $\langle \mathcal{M}, C \rangle$ of a transition structure $\mathcal{M}$ and a set C of fairness constraints. Paths of $\langle \mathcal{M}, C \rangle$ are paths of $\mathcal{M}$ that satisfy all fairness constraints $c \in C$.

Definition 9.1.1. c is a *simple fairness constraint* if c is a CTL (state) formula.

Recall that a state formula is also a path formula. Each state formula c characterises a set S_c of states. A simple fairness constraint c expresses that paths should pass infinitely often through states of S_c.

Definition 9.1.2. The paths of a system $\langle \mathcal{M}, C \rangle$ with C a set of simple fairness constraints are all paths of $\mathcal{M}$ that pass infinitely often through states of S_c, for every $c \in C$.

To implement model checking in LTL and CTL*, it suffices to express fairness constraints as a formulas and apply the existing model checking algorithms. As seen before, quantification over

fair paths can be expressed in CTL*, using $\mathrm{A}((\mathrm{G}\,\mathrm{F}\,c_1) \wedge \cdots \wedge (\mathrm{G}\,\mathrm{F}\,c_n) \Rightarrow \psi)$, and $\mathrm{E}((\mathrm{G}\,\mathrm{F}\,c_1) \wedge \cdots \wedge (\mathrm{G}\,\mathrm{F}\,c_n) \wedge \psi)$. Thus, there is no need to extend model checking algorithms for LTL and CTL* to handle fairness.

For CTL formulas φ, the situation is different. The resulting formula is not a CTL formula. A different approach is followed.

Extending the CTL* with fairness quantifiers Given a set C of fairness constraints, we extend CTL* with path quantifiers $\mathrm{A}_C, \mathrm{E}_C$. They quantify over fair paths w.r.t. C.

Definition 9.1.3. We extend the standard state formula satisfaction relation with two additional rules:

- $\mathcal{M}, s \models \mathrm{A}_C[\psi]$ if for each path π of $\mathcal{M}$ fair to C, $\mathcal{M}, \pi \models \psi$.
- $\mathcal{M}, s \models \mathrm{E}_C[\psi]$ if for some path π of $\mathcal{M}$ fair to C, $\mathcal{M}, \pi \models \psi$.

Given a system $\langle \mathcal{M}, C \rangle$, and φ a CTL* formula, let φ_C be the formula obtained by transforming A into A_C and E into E_C.

Proposition 9.1.2. $\langle \mathcal{M}, C \rangle, s \models \varphi$ *iff* $\mathcal{M}, s \models \varphi_C$.

CTL with fairness quantifiers The CTL* quantifiers $\mathrm{A}_C, \mathrm{E}_C$ induce the following additional CTL connectives with fairness:

$$\mathrm{A}_C\,X, \mathrm{E}_C\,X, \mathrm{A}_C\,F, \mathrm{E}_C\,F, \mathrm{A}_C\,G, \mathrm{E}_C\,G, A_C U, E_C U$$

One can show as before that the following is an adequate set:

$$\mathrm{E}_C\,X, \mathrm{E}_C\,G, \mathrm{E}_C\,U, \wedge, \neg, \bot$$

Moreover, the following proposition shows that $\mathrm{E}_C\,X$ and $\mathrm{E}_C\,\mathrm{U}$ can be expressed in terms of EX, E U and $\mathrm{E}_C\,\mathrm{G}$.

Proposition 9.1.3.

- $\mathrm{E}_C[\phi\,\mathrm{U}\,\psi] \equiv \mathrm{E}[\phi\,\mathrm{U}(\psi \wedge \mathrm{E}_C\,\mathrm{G}\,\top)]$
- $\mathrm{E}_C\,X\psi \equiv \mathrm{EX}(\psi \wedge \mathrm{E}_C\,\mathrm{G}\,\top)$

The proof is based on the fact that a fairness property is a property of the tail of a path. For any tail of a path, it holds that the path is fair iff this tail is fair. Hence, there is a fair path satisfying $\phi\,\mathrm{U}\,\psi$ if there is a finite path to a state satisfying ψ that can be extended to a path that is fair with respect to C.

Combining the above results, we find that a decision problem $\langle \mathcal{M}, C \rangle, s \models \varphi$ can be reduced to the problem $\mathcal{M}, s \models \varphi'$ where φ' uses only standard CTL operators and $\mathrm{E}_C\,\mathrm{G}$.

An algorithm for fair model checking in CTL To implement CTL model checking for systems with simple fairness constraints, it suffices to extend the CTL model checking algorithm with one operator $\mathrm{E}_C\,\mathrm{G}$.

As seen before, every CTL formula with respect to a system with fairness constraints C can be transformed into a formula using EX, EU and one fairness connective $\mathrm{E}_C\,\mathrm{G}$. Hence, to adapt the model checking algorithm for fairness, it suffices to extend the procedure for computing $S_{\mathrm{E}_C\,\mathrm{G}\,\phi}$.

To compute $S_{\mathrm{E}_C\,\mathrm{G}\,\phi}$ we use a variant of the linear algorithm.

In a prior step to model checking in $\langle \mathcal{M}, c\rangle$, we compute S_c, for every $c \in C$. Since c is a CTL formula, the standard algorithm can be used to compute S_c. This need to be done only once for $\langle \mathcal{M}, c\rangle$.

In the recursive model checking algorithm, the following computation step is plugged in for computing $S_{\mathrm{E}_C\,\mathrm{G}\,\phi}$.

- Compute $\mathcal{M}'$ by restricting S to S_ϕ (which has been recursively computed). That is, delete states in which ϕ is false and their edges.
- Compute all maximal SCC's (strongly connected components) of $\mathcal{M}'$ using Tarjans algorithm. This is as before.
- Compute the subset of "fair" such components: a maximal SCC is fair to C if for every $c \in C$, it holds that $SCC \cap S_c \neq \emptyset$. I.e., some state in SSC satisfies the state formula c.

  In a SCC, there exists a loop that passes over all states of it. It is obvious then that each finite path in $\mathcal{M}'$ that reaches a "good" SCC can be extended into a path that is fair to C.
- Finally, compute $S_{E_C G\phi}$ as the set of states in $\mathcal{M}'$ that can reach a fair strongly connected component.

This procedure can be performed in $O(V+E)$. Total complexity of the model checking algorithm with fairness is $O(n \times f \times (V+E))$, with n the number of fairness constraints. This algorithm is linear in the size of $\mathcal{M}$.

Exercise 9.1.3. *Given an argument for the correctness of the modified algorithm. In particular, explain how to construct for each state in $S_{E_C G\phi}$ a fair path on which ϕ is globally true.*

9.2 An LTL model checking algorithm

In this section, we develop an algorithm implementing the following inference.

LTL modelchecking:

- Input: a system $\langle \mathcal{M}, C\rangle$ with C consisting of simple fairness constraints, $s_0 \in S$ and LTL formula Φ.
- output: $(\mathcal{M}, s_0 \models \mathrm{A}_C\,\Phi)$.

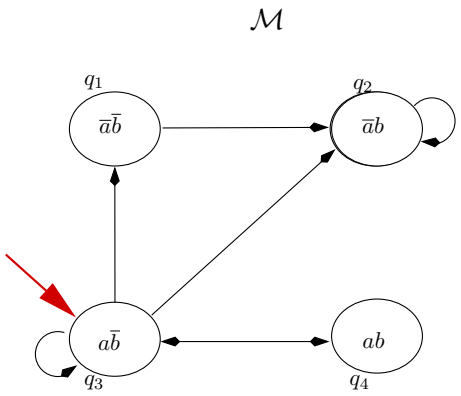


Figure 9.1: An LTL-model checking problem

Recall that a simple fairness constraint is a path constraint characterized by a CTL formula φ. It is the constraint that a path should pass infinitely often through states s such that $\mathcal{M}, s \models \varphi$.

The inference problem to be solved is to verify whether $\pi \models \Phi$ for each C-fair path $\pi = s_0 \rightarrow \ldots$.

Notice that $\langle \mathcal{M}, C \rangle, s_0 \models \mathrm{A}\,\Phi$ iff $\mathcal{M}, s_0 \models \mathrm{A}_C\,\Phi$. Since an LTL formula Φ does not contain path quantifiers.

The adequate set We solve the inference problem for LTL formulas using the adequate set of LTL connectives $\{\neg, \wedge, \mathrm{X}, \mathrm{U}\}$. Recall that $\mathrm{F}\,\alpha \equiv \top \,\mathrm{U}\, \alpha; \mathrm{G}\,\alpha \equiv \neg\,\mathrm{F}\,\neg\alpha, \ldots$.

LTL model checking: the idea The algorithm presented below was developed by Vardi & Wolper in 1984 and uses techniques from automata theory. The strategy of the algorithm is to search for a C-fair path π in $\mathcal{M}$ such that $\pi \models \neg\Phi$. If it succeeds, then $\mathcal{M}, s_0 \not\models \mathrm{A}_C\,\Phi$. If it fails, then $\mathcal{M}, s_0 \models \mathrm{A}_C\,\Phi$.

Example 9.2.1. As a running example, consider the LTL model checking problem in Figure 9.1. The problem is to decide $\mathcal{M}, q_3 \models_C \neg(a \,\mathrm{U}\, b)$. Here the set of fairness constraints C is empty. The answer to this decision problem is **f**: $\mathcal{M}, q_3 \not\models \mathrm{A}\,\neg(a \,\mathrm{U}\, b)$. Evidence is provided by paths satisfying $a \,\mathrm{U}\, b$ such as $(q_3 \rightarrow)^* \rightarrow q_4 \rightarrow \ldots$ and $(q_3 \rightarrow)^* \rightarrow q_2 \rightarrow \ldots$.

LTL model checking: the intuition Let $Sub(\Phi)$ denote the set of all subformulas of Φ and the negations of these subformulas.

We define the *LTL formula set at i* of path π as the set $Q_i = \{\varphi \in Sub(\Phi) \mid \pi^i \models \varphi\}$. We define the *annotated path of* π as the sequence:

$$\begin{pmatrix} Q_0 \\ s_0 \end{pmatrix} \rightarrow \ldots \rightarrow \begin{pmatrix} Q_n \\ s_n \end{pmatrix} \rightarrow \ldots$$

where Q_i is the LTL formula set of π at i. Each path π determines one unique annotated path.

The strategy of the algorithm is to compute a new transition structure $\mathcal{M}^\times$ and a set $C^\times$ of fairness constraints, such that:

- The states of $\mathcal{M}^\times$ are of the form $\begin{pmatrix} Q \\ s \end{pmatrix}$ with s a state of $\mathcal{M}$ and $Q \subseteq Sub(\Phi)$;

- The set of $C^\times$-fair paths $\pi' = \begin{pmatrix} Q_0 \\ s_0 \end{pmatrix} \to \ldots \to \begin{pmatrix} Q_n \\ s_n \end{pmatrix} \to \ldots$ of $\mathcal{M}^\times$ is exactly the set of annotated C-fair paths $\pi = s_0 \to \ldots \to s_n \to \ldots$ of $\mathcal{M}$.

With $\mathcal{M}^\times, C^\times$ we are then able to solve our problem $\mathcal{M}, s_0 \models \mathrm{A}_C\, \Phi$. Indeed, $\mathcal{M}, s_0 \not\models \mathrm{A}_C\, \Phi$ iff there is a C-fair path $\pi = s_0 \to \ldots$ in $\mathcal{M}$ such that $\mathcal{M}, \pi \models \neg\Phi$ iff there is a $C^\times$-fair path $\pi' = (s_0, Q_0) \to \ldots$ of $\mathcal{M}^\times$ such that $\neg\Phi \in Q_0$.

An algorithmic solution to the problem is to compute $S_{\mathrm{E}_{C^\times}\, \mathrm{G}\, \top}$ in $\mathcal{M}^\times$, and to check if it contains a state (s_0, Q_0) with $\neg\Phi \in Q_0$.

We have reduced the LTL model checking problem in $\langle \mathcal{M}, C \rangle$ to CTL model checking in $\langle \mathcal{M}^\times, C^\times \rangle$ and $C^\times$.

We now discuss how to construct $\mathcal{M}^\times = \langle S^\times, \to^\times, L \rangle$.

Construction of $S^\times$. The product state space $S^\times$ consists of certain pairs (s, Q) with $s \in S$ and $Q \subseteq Sub(\Phi)$.

To reduce search space, $S^\times$ is as small as possible. This is done by eliminating pairs (s, Q) that cannot occur in an annotated path; i.e., pairs (s, Q) for which no path $\pi = s \to \ldots$ can exist such that $Q = \{\varphi \in Sub(\Phi) \mid \pi \models \varphi\}$.

There are two types of inconsistent pairs (s, Q). One is where Q is a logically inconsistent set. The second one is where s and Q disagree on the value of some propositional symbol p. This is explained below.

Let π be a path and Q the set of elements of $Sub(\Phi)$ such that $\pi \models \psi$. I.e., $Q = \{\psi \in Sub(\Phi) \mid \pi \models \psi\}$. Q can be seen to satisfy certain consistency conditions.

- $\pi \models \gamma$ or $\pi \models \neg\gamma$. Hence, for each subformula γ of Φ, either $\gamma \in Q$ or $\neg\gamma \in Q$.
- It holds that $\pi \models \gamma \wedge \delta$ iff $\pi \models \gamma$ and $\pi \models \delta$. Therefore, Q contains $\gamma \wedge \delta$ iff it contains γ and δ.
- If $\pi \models \gamma\, \mathrm{U}\, \delta$, then $\pi \models \gamma$ or $\pi \models \delta$. Hence, if Q contains $\gamma\, \mathrm{U}\, \delta$, then it contains γ or δ.
- If $\pi \models \neg(\gamma\, \mathrm{U}\, \delta)$, then $\pi \models \neg\delta$. Hence, if $\neg(\gamma\, \mathrm{U}\, \delta) \in Q$ then $\neg\delta \in Q$.

These are the conditions for the consistency of an LTL formula sets Q.

Closed formula sets Let $Sub(\Phi)$ denote the set of all subformulas of Φ and the negations of these subformulas.

Definition 9.2.1. A *closed formula set* $Q \subseteq Sub(\Phi)$ of Φ satisfies:

- It contains for each subformula γ of Φ either γ or $\neg\gamma$.
- For each $\gamma \wedge \delta \in Sub(\Phi)$, $\gamma \wedge \delta \in Q$ iff $\gamma \in Q$ and $\delta \in Q$.
- If $\gamma\, \mathrm{U}\, \delta \in Q$ then $\gamma \in Q$ or $\delta \in Q$.
- If $\neg(\gamma\, \mathrm{U}\, \delta) \in Q$ then $\neg\delta \in Q$.

Denote the set of closed formula sets of Φ by $\mathcal{C}(\Phi)$.

Observe that each closed formula set contains either p or $\neg p$, for each symbol p in Φ.

Definition 9.2.2. We call a state s of $\mathcal{M}$ *consistent* with a closed formula set Q if for each atom p that appears in Φ, $p \in L(s)$ iff $p \in Q$. We denote this by $s /\!\!/ Q$.

If s is not *consistent* with Q then Q contains the negation of a literal that is true in s. Hence, no path in s can satisfy all formulas in Q.

Definition 9.2.3. Define $S^{\times} := \{(s, Q) | s \in S \wedge Q \in \mathcal{C}(\Phi) \wedge s /\!\!/ Q\}$

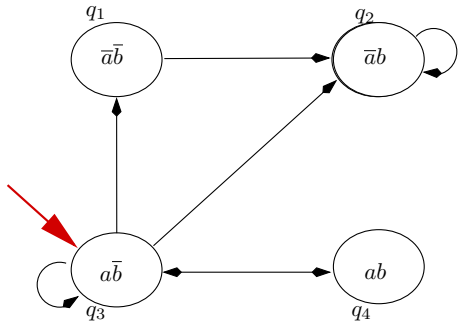
Example 9.2.2. The running example: closed formula sets. The closed formula sets are the maximal consistent subsets of $\{a, \neg a, b, \neg b, a \,\mathrm{U}\, b, \neg(a \,\mathrm{U}\, b), \neg\neg(a \,\mathrm{U}\, b)\}$.

We use a compact notation for these sets based on the following notations. We denote $\alpha := (a \,\mathrm{U}\, b)$. We use $\overline{\delta}$ to denote $\neg\delta$. A closed formula set contains $\neg\neg(a \,\mathrm{U}\, b)$ iff it does not contain $\neg(a \,\mathrm{U}\, b)$ iff it contains $a \,\mathrm{U}\, b$. We do not write $\neg\neg(a \,\mathrm{U}\, b)$. It is understood that $\neg\neg(a \,\mathrm{U}\, b)$ belongs to a set if $a \,\mathrm{U}\, b$ does.

E.g., $\overline{a}b\overline{\alpha}$ corresponds to $\{\neg a, b, \neg(a \,\mathrm{U}\, b)\}$. This is not a closed formula set due to b and $\neg(a \,\mathrm{U}\, b)$. E.g., $ab\alpha$ corresponds to $\{a, b, a \,\mathrm{U}\, b, \neg\neg(a \,\mathrm{U}\, b)\}$. This is a closed formula set.

The closed formula sets correspond to all combinations of $\{a, \overline{a}\} \times \{b, \overline{b}\} \times \{\alpha, \overline{\alpha}\}$ except $\overline{a}\overline{b}\alpha$ and $ab\overline{\alpha}$ and $\overline{a}b\overline{\alpha}$. That is, $\mathcal{C}(\Phi) = \{ab\alpha, a\overline{b}\alpha, a\overline{b}\overline{\alpha}, \overline{a}b\alpha, \overline{a}\overline{b}\overline{\alpha}\}$

Example 9.2.3. The running example: $S^{\times}$



$$\mathcal{C}(\Phi) = \{\ \overline{a}\overline{b}\overline{\alpha}, \overline{a}b\alpha, a\overline{b}\alpha, a\overline{b}\overline{\alpha}, ab\alpha\ \}$$

What are the consistent pairs?

- $q_1 /\!\!/ : \overline{a}\overline{b}\overline{\alpha}$
- $q_2 /\!\!/ : \overline{a}b\alpha$
- $q_3 /\!\!/ : a\overline{b}\overline{\alpha}, a\overline{b}\alpha$
- $q_4 /\!\!/ : ab\alpha$

$$S^{\times} = \{(q_1, \overline{a}\overline{b}\overline{\alpha}), (q_2, \overline{a}b\alpha), (q_3, a\overline{b}\overline{\alpha}), (q_3, a\overline{b}\alpha), (q_4, ab\alpha)\}$$

We denote these pairs more compactly as

$$S^{\times} = \{q_1, q_2, q_3', q_3, q_4\}$$

The states q_1, q_2, q_4 correspond to unique states of $S^\times$. q_3 is the only state that was duplicated. Why? Because each of q_1, q_2, q_4 trivially determines whether α is true or not in a path starting in it, and q_3 does not.

In general, if Φ is large, then each state s might be duplicated many times. Vice versa, each closed formula set Q might be consistent with many or with no states $s \in S$, and hence, might be duplicated many times or may not occur at all in $S^\times$. But such situations do not arise in the running example.

Exercise 9.2.1. *Think of an example where state sets get duplicated.*

Construction of $\to^\times$. We design $\to^\times$ as small as possible such that if $(s, Q) \to^\times (s', Q')$, then such a transition could occur in an annotated path. Cases of impossible transitions are:

- s' is not a possible successor of s: $s \not\to s'$
- Q' is not a possible successor of Q:
  - $\mathrm{X}\, p \in Q$ and $\neg p \in Q'$
  - $\neg \mathrm{X}\, p \in Q$ and $p \in Q'$
  - $p \,\mathrm{U}\, q \in Q$, $\neg q \in Q$ but $\neg(p \,\mathrm{U}\, q) \in Q'$
  - $\neg(p \,\mathrm{U}\, q) \in Q$, $p \in Q$ but $p \,\mathrm{U}\, q \in Q'$

Exercise 9.2.2. *Explain.*

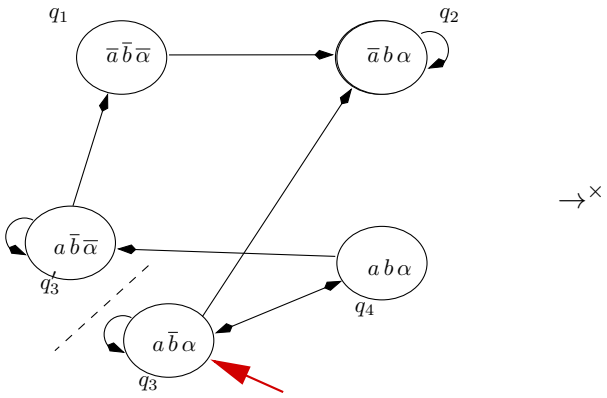
Definition 9.2.4. For all $(s, Q), (s', Q') \in S^\times$, there is a transition $(s, Q) \to^\times (s', Q')$ if all following conditions hold:

- $s \to s'$ (consistency with $\mathcal{M}$)
- Q' is a consistent successor of Q:
  - If $\mathrm{X}\, \delta \in Q$ then $\delta \in Q'$.
  - If $\neg \mathrm{X}\, \delta \in Q$ then $\neg\delta \in Q'$.
  - If $\gamma \,\mathrm{U}\, \delta \in Q$ and $\neg\delta \in Q$, then $\gamma \,\mathrm{U}\, \delta \in Q'$.
  - If $\neg(\gamma \,\mathrm{U}\, \delta) \in Q$ and $\gamma \in Q$ then $\neg(\gamma \,\mathrm{U}\, \delta) \in Q'$.

Example 9.2.4. The running example: $\to^\times$ in $S^\times := \{(q_1, \overline{a}\overline{b}\overline{\alpha}), (q_2, \overline{a}b\alpha), (q_3, a\overline{b}\overline{\alpha}), (q_3, a\overline{b}\alpha), (q_4, ab\alpha)\}$ $\to^\times$ is:

- $(q_1 =)(q_1, \overline{a}\overline{b}\overline{\alpha}) \to^\times (q_2, \overline{a}b\alpha)$
- $(q_2 =)(q_2, \overline{a}b\alpha) \to^\times (q_2, \overline{a}b\alpha)$
- $(q_3' =)(q_3, a\overline{b}\overline{\alpha}) \to^\times (q_1, \overline{a}\overline{b}\overline{\alpha}), (q_3, a\overline{b}\overline{\alpha})(= q_3')$
- $(q_3 =)(q_3, a\overline{b}\alpha) \to^\times (q_2, \overline{a}b\alpha), (q_4, ab\alpha), (q_3, a\overline{b}\alpha)(= q_3)$

- $(q_4 =)(q_4, ab\alpha) \to^\times (q_3, a\bar{b}\alpha)(= q_3), (q_3, a\bar{b}\overline{\alpha})(= q_3')$



Are we done? Unfortunately not.

Recall, we are designing $\mathcal{M}^\times$ such that its paths are annotated paths of $\mathcal{M}$. In particular, it should be the case that if $(s, Q) \to^\times \dots$ is a path of $\mathcal{M}^\times$, then $Q = \{\varphi \in Sub(\Phi) \mid (s \to \dots) \models \varphi\}$.

The transition relation $\to^\times$ in Example 9.2.4 still has paths $\pi = (s, Q) \to \dots$ that are not annotated paths of $\mathcal{M}$, that is such that for some $\psi \in Q$, it holds that $\pi \not\models \psi$.

One such a path is the loop $\pi = (q_3, a\bar{b}\alpha) \to (q_3, a\bar{b}\alpha) \to \dots$. It holds that $\pi \models a \,\mathrm{W}\, b$ but $\pi \not\models a \,\mathrm{U}\, b$.

To eliminate this path, we should forbid paths with an infinite tail containing $a \,\mathrm{U}\, b, a, \neg b$. How? By using a fairness condition!

Fairness conditions $C^\times$ A fairness condition is specified here as a set c of states. It expresses that a fair path $\pi = s_0 \to \dots$ should visit infinitely often an element of c. That is, for infinitely many i, $s_i \in c$.

There are two sources of fairness constraints: $C^\times = C^{\mathrm{U}} \cup C'$.

One set of fairness constraints C^{U}. Their source are the until statements in $Sub(\Phi)$. No path π should have a tail π^i in which two formulas $\gamma \,\mathrm{U}\, \delta$ and $\neg\delta$ belong to Q_j for all $j \geq i$. Notice that if $\gamma \,\mathrm{U}\, \delta \in Q_j, \neg\delta \in Q_j$ then since Q_j is a closed formula set, it holds that $\gamma \in Q_j$. Hence, this is a path in which $\gamma \,\mathrm{W}\, \delta$ is true but not $\gamma \,\mathrm{U}\, \delta$ then the precondition γ remains true forever and the postcondition δ never becomes true.

Equivalently, any path should visit infinitely often the set of states $c^{\gamma \,\mathrm{U}\, \delta} = \{(s, Q) \in S^\times \mid \neg(\gamma \,\mathrm{U}\, \delta) \in Q \text{ or } \delta \in Q\}$.

Definition 9.2.5. Define $C^{\mathrm{U}} = \{c^{\gamma \,\mathrm{U}\, \delta} \mid \gamma \,\mathrm{U}\, \delta \in Sub(\Phi)\}$.

Another source of fairness constraints is C, the original set of (simple) fairness constraints of the LTL model checking problem. A simple fairness constraint is a path constraint characterized by a CTL formula φ_c, or equivalently, by the set $S_{\varphi_c} = \{s \in S | \mathcal{M}, s \models \varphi_c\}$ of states of $\mathcal{M}$ satisfying

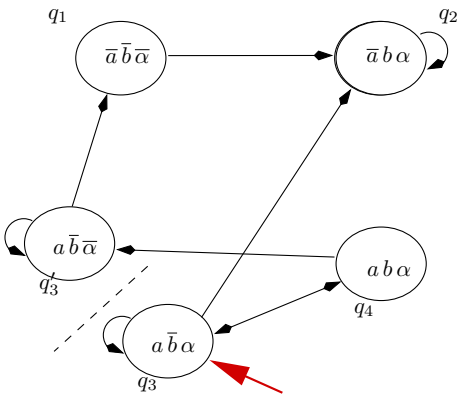
φ_c. The fairness path constraint is that the path should pass infinitely often through states of S_{φ_c}.

c is to be translated into a fairness constraint for $S^\times$: $c^\times = \{(s, Q) \in S^\times | s \in S_{\varphi_c}\}$.

Definition 9.2.6. Define $C' = \{c^\times \mid c \in C\}$.

Definition 9.2.7. Define $C^\times = \{c^\times | c \in C\} \cup \{c^{\gamma \,\mathrm{U}\, \delta} | \gamma \,\mathrm{U}\, \delta \in Sub(\Phi)\}$

Example 9.2.5. The running example: fairness conditions $C^\times$



$$\begin{aligned} C &= \emptyset \\ C^\times &= \{c^{a \,\mathrm{U}\, b}\} \\ &= \{\{q_1, q_2, q_3', q_4\}\} \\ &= \{S^\times \setminus \{q_3\}\} \end{aligned}$$

The unfair paths of $\mathcal{M}^\times$ have a tail $(q_3 \to^\times)^\infty$. E.g. $(q_3 \to^\times)^\infty$ and $q_3 \to^\times q_4 \to^\times (q_3 \to^\times)^\infty$.

No path of $\mathcal{M}$ corresponds to such unfair paths. Indeed, such an unfair path would correspond to a path in $\mathcal{M}$ with tail $(q_3 \to)^\infty$. But $\alpha = a \,\mathrm{U}\, b$ is not satisfied in this tail.

The path $(q_3 \to)^\infty$ of $\mathcal{M}$ matches only the fair path $(q_3' \to^\times)^\infty$.

There are infinitely many fair paths of $\mathcal{M}^\times$ in q_3. E.g. $q_3 \to^\times q_4 \to^\times q_3' \to^\times q_1(\to^\times q_2)^\infty$ matches with $q_3 \to q_4 \to q_3 \to q_1(\to q_2)^\infty$ of $\mathcal{M}$.

Example 9.2.6. Summary of the running example. We verify that $\mathcal{M}, q_3 \not\models \Phi$ where $\Phi = \neg(a \,\mathrm{U}\, b)$. Indeed, the state q_3 in the constructed transition graph $\mathcal{M}^\times$ has a fair path, e.g., $q_3 \to q_4 \to \ldots$. The state q_3 in $\mathcal{M}^\times$ stood for the pair $(q_3, a\bar{b}\alpha)$ where $a\bar{b}\alpha$ is itself a shorthand notation for a closed formula set that contains $\neg\Phi = \neg(\neg(a \,\mathrm{U}\, b))$. This proves that $\mathcal{M}, q_3 \not\models \Phi$.

Theorem 9.2.1. *A path π' is a C^{U}-fair path of $\mathcal{M}^\times$ iff π' is an annotated path of $\mathcal{M}$. A path π' is a $C^\times$-fair path of $\mathcal{M}^\times$ iff π' is the annotated path of a C-fair path of $\mathcal{M}$.*

Summary and complexity

Theorem 9.2.2. *$\mathcal{M}, s_0 \models \mathrm{A}_C\,\Phi$ iff $M^\times$ has no $C^\times$-fair path $(s_0, Q) \to \ldots$ such that $\neg\Phi \in Q$.*

Exercise 9.2.3. *Prove that this follows from the previous theorem.*

The LTL model checking algorithm is called to decide $\mathcal{M}, s_0 \models_C \Phi$. It solves this problem as follows:

- Compute $S^\times, \to^\times, C^\times$.
- Compute the maximal strongly connected components SCC of $\to^\times$ that are fair to $C^\times$. Such a SCC has the property that $SSC \cap c \neq \emptyset$, for each $c \in C^\times$.
- If some $(s_0, Q) \in S^\times$ with $\neg\Phi \in Q$ has a path to a fair strongly connected component, then return **f** else return **t**.

Complexity of the algorithm Exploiting results for CTL-model checking, we see that this algorithm is exponential in the size of Φ. Since the number of closed formula sets is exponential in the size of Φ. However, it is linear in the size of $\mathcal{M}$.

LTL-model checking is harder than CTL-checking because the construction of $\mathcal{M}^\times$ blows up the transition system.

9.2.1 Summary of the method

Definition 9.2.8. LTL model-checking is the inference problem that takes as input:

- Σ : propositional vocabulary
- $\mathcal{M} = \langle S, \to, L\rangle$, with $L : S \to 2^\Sigma$
- C: set of CTL formulas over Σ
- $s_0 \in S$
- Φ: LTL formula over Σ

and returns $(\mathcal{M}, s_0 \models \mathrm{A}_C\,\Phi)$.

Recall that $\{\neg, \wedge, \mathrm{X}, \mathrm{U}\}$ is an adequate set of LTL. We assume that Φ uses only the LTL connectives from this set.

Definition 9.2.9. $Sub(\Phi) = \{\varphi, \neg\varphi \mid \varphi \text{ is a subformula of } \Phi\}$

Definition 9.2.10. A set $Q \subseteq Sub(\Phi)$ is a closed formula set of Φ if:

- For each $\gamma \in Sub(\Phi)$ Q contains either γ or $\neg\gamma$.
- For each $\gamma \wedge \delta \in Sub(\Phi)$, $\gamma \wedge \delta \in Q$ iff $\gamma \in Q$ and $\delta \in Q$.
- If $\gamma \,\mathrm{U}\, \delta \in Q$ then $\gamma \in Q$ or $\delta \in Q$.
- If $\neg(\gamma \,\mathrm{U}\, \delta) \in Q$ then $\neg\delta \in Q$.

Definition 9.2.11. $\mathcal{C}(\Phi)$ is the set of closed formula sets of Φ.

Definition 9.2.12. $s /\!\!/ Q$ if $s \in S, Q \in \mathcal{C}(\Phi)$ and for each atom p that appears in Φ, $p \in L(s)$ iff $p \in Q$.

Definition 9.2.13. $S^{\times} := \{(s, Q) | s \in S \wedge Q \in \mathcal{C}(\Phi) \wedge s /\!\!/ Q\}$

Definition 9.2.14. $(s, Q) \rightarrow^{\times} (s', Q')$ if all following conditions hold:

- $(s, Q), (s', Q') \in S^{\times}$
- $s \rightarrow s'$ (consistency with $\mathcal{M}$)
- Q' is a consistent successor of Q:
  - If $\mathrm{X}\, \delta \in Q$ then $\delta \in Q'$.
  - If $\neg \mathrm{X}\, \delta \in Q$ then $\neg\delta \in Q'$.
  - If $\gamma \,\mathrm{U}\, \delta \in Q$ and if $\neg\delta \in Q$, then $\gamma \,\mathrm{U}\, \delta \in Q'$.
  - If $\neg(\gamma \,\mathrm{U}\, \delta) \in Q$ and if $\gamma \in Q$ then $\neg(\gamma \,\mathrm{U}\, \delta) \in Q'$.

Definition 9.2.15. $c^{\gamma \,\mathrm{U}\, \delta} = \{(s, Q) \in S^{\times} \mid \neg(\gamma \,\mathrm{U}\, \delta) \in Q \vee \delta \in Q\}$

Definition 9.2.16. For any $c \in C$, define $c^{\times} = \{(s, Q) \in S^{\times} | s \in S_c\}$

Definition 9.2.17. Define $C^\times = \{c^\times | c \in C\} \cup \{c^{\gamma \,\mathrm{U}\, \delta} | \gamma \,\mathrm{U}\, \delta \in Sub(\Phi)\}$

Correctness theorem

Theorem 9.2.3. *$\mathcal{M}, s_0 \models \mathrm{A}_C\, \Phi$ iff there is no path through $\to^\times$ starting in a state $(s_0, Q) \in S^\times$ with $\neg\Phi \in Q$ that is $C^\times$-fair.*

Algorithm Algorithm: Is $\mathcal{M}, s_0 \models_C \Phi$?

- Compute $S^\times, \to^\times, C^\times$.
- Compute the strongly connected components of $\to^\times$
- Delete all the strongly connected components S' such that $S' \cap c = \emptyset$, for some $c \in C^\times$.
- If some $(s_0, Q) \in S^\times$ with $\neg\Phi \in Q$ has a path to one of the remaining strongly connected components thenanswer **f** else **t**.

From the second step on, this is the algorithm for CTL model checking applied to compute $S_{\mathrm{E}_{C^\times}\, \mathrm{G}\, \top}$.

Complexity of the algorithm The algorithm is exponential in the size of Φ since the number of closed formula sets is exponential in the size of Φ. The algorithm is linear in the size of $\mathcal{M}$.

LTL-model checking is harder than CTL-checking because composition blows up the transition system.

9.3 Important for exam

```
Big questions:
- CTL model checking algorithm  with fairness
- LTL model checking
```

Bibliography

[1] R. Back, "On correct refinement of programs," *J. Comput. Syst. Sci.*, vol. 23, no. 1, pp. 49–68, 1981.

[2] J. Abrial, *Modeling in Event-B - System and Software Engineering.* Cambridge University Press, 2010.

[3] D. Cansell and D. Méry, "Foundations of the B method," *Computers and Artificial Intelligence*, vol. 22, no. 3-4, pp. 221–256, 2003.