MCS-summary

Kevin Sun

June 15, 2020

Contents

1 Chapter 2

1.1 Derived semantical concepts: formal definitions

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.* Otherwise, a theory contains *incomplete knowledge.*

1.2 Inference problems

- The evaluation inference problem:
  - 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 **t** if e is satisfied in $\mathfrak{A}$ and **f** otherwise.
- The validity checking inference problem:
  - Input: theory T
  - Output: $e^{\mathfrak{A}}$

  Logically valid means tautological.
- The satisfiability checking inference problem:
  - Input: theory T
  - Output: **t** if T is satisfiable, **f** otherwise
- The model generation inference problem:
  - Input: theory T
  - Output: the set of models of T. This is the set $\{\mathfrak{A} \mid \mathfrak{A} \models T\}$.
- The deduction inference problem:
  - Input: theory T, boolean expression e
  - Output: **t** if T logically entails e, **f** otherwise

  Theory T entails expression e if every structure $\mathfrak{A}$ that satisfies T satisfies e as well. Formally, $\{\mathfrak{A} \mid \mathfrak{A} \models T\} \subseteq \{\mathfrak{A} \mid \mathfrak{A} \models e\}$

2 Chapter 4

2.1 Lightweight verification by finite model generation

The verification problem This problem consists of verifying if a property is entailed by the formal specification (an *emergent* property). A kind of property 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.

However, proving such an emergent property is a non-trivial deductive inference problem. For this reason, Alloy focuses on simpler forms of reasoning based on finite model generation. It does not 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 the states are at different time points.

2.2 LTC modelling of programs

A program is a specification of dynamic processes. 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.

2.2.1 Applications of lightweight verification of programs

Simultaneously look at the running example introduced at the bottom of page 121 in the course notes for a more clear read.

Lightweight 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. Generating a test value for a given chosen control flow is done by generating a constraint on the execution flow. *CurPP(t)* is a function that returns the current program point. The following expresses that at point 2 it must go along P4 and at point 5 it must go along P3 (it is implicit that P3 and P4 are e.g. different clauses in an if-statement.)

$$CurPP(2) = P4 \wedge CurPP(5) = P3$$

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.

2.3 Heavyweight verification of invariants

To verify an invariant , one must verify that the theory T satisfies the invariant at every point in time. This could be done using a theorem prover. However T contains the interpreted type $\mathtt{T}$ interpreted by $\mathbb{N}$. Thus, theorem provers should be able to reason about the natural numbers. This means they must have a theory of natural numbers. Peano's FO arithmetic must be used as no good automated SO provers exist. We will use the induction schema $Ind(\varphi[t])$ to prove invariants $\varphi[t]$.

$$Ind(\varphi[t]) \equiv \varphi[0] \wedge \forall t(\varphi[t] \Rightarrow \varphi[S(t)]) \Rightarrow \forall t\varphi[t]$$

This method is applicable to bi-state LTC theories.

Definition 2.1. An LTC theory is 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 2.2. 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.

2.3.1 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 $\psi = \varphi[t]$
- First, we construct the following theories:

  - $T_{init} = T_{static} \cup T_0 \cup T_s[0]$
  - $T_{ind} = T_{static} \cup T_s[0] \cup T_s[1] \cup T_t[0] \cup \{\varphi[0]\}$

- We use a theorem prover to verify:
  - $T_{init} \models \varphi[0]$
  - $T_{ind} \models \varphi[1]$

- If both calls succeed, then return that $\varphi[t]$ is an invariant of T_a, otherwise report failure.

If $T_{init} \models \varphi[0]$, then also $T_a \models \varphi[0]$, since $T_a \models T_i nit$. This establishes the base case of the induction proof.

T_{ind} relates any two successive poitins in time, as it has no initial state information (T_0 is not present in the definition of T_{ind}). Therefore, if $T_{ind} \models \varphi[1]$, then it follows that:

$$T_a \models \forall t(\varphi[t] \Rightarrow \varphi[S(t)]).$$

This establishes the induction step of the proof.

If both calls to the theorem prover succeed, it holds that:

- $T_a \models \varphi[0]$
- $T_a \models \forall t(\varphi[t] \Rightarrow \varphi[S(t)])$

Combining this with $Ind(\varphi[t])$, we can conclude that:

$$T_a \models \forall t \varphi(t)$$

Proof of method:

Theorem 1. If $T_{init} \models \varphi[0]$ and $T_{ind} \models \varphi[1]$, then $T_a \models \forall t \varphi[t]$

Proof. $Ind(\varphi[t]) := \varphi[0] \wedge \forall t(\varphi[t] \Rightarrow \varphi[S(t)]) \Rightarrow \forall t \varphi[t]$

$T_a \models Ind(\varphi[t])$, because `T` is the interpreted type $\mathbb{N}$ for which all induction schema instances hold.

$T_a \models \varphi[0]$, because $T_{init} \models \varphi[0]$ and $T_a \models T_{init}$

Since $T_{ind} \models \varphi[1]$, the following holds:

- Then $T_{static} \cup T_s[0] \cup T_s[S(0)] \cup T_t[0] \cup \{\varphi[0]\} \models \varphi[S(0)]$

  (Use of definition of T_{ind} and 1 can be expressed as the successor of 0, $1 = S(0)$)

- Then $T_{static} \cup T_s \cup T_t \models \varphi[0] \Rightarrow \varphi[S(0)]$

  (The theory $T[n]$ is a subset of the theory T it is based on. Anything entailed by $T[n]$ is also entailed by T.)

- Then $T_{static} \cup T_s \cup T_t \models \forall t(\varphi[t] \Rightarrow \varphi[S(t)])$

  (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 $T_a \models \forall t(\varphi[t] \Rightarrow \varphi[S(t)])$.

  ($T_a \models T_{static} \cup T_s \cup T_t$)

The application of $Ind(\varphi[t])$ yields that $T_a \models \forall t\varphi[t]$.

2.3.2 Limitations of the verification method

Incomplete method The method is not complete. There are two cases where it may fail to prove the invariance of an invariant:

- If the invariants are not sufficiently strong: 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. This means the method reports a failure, but the property could be a still be an invariant.
- 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.

2.4 Progression inference

Definition 2.3. 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.

A finite sequence S of these structures correspond to a structure that includes time arguments bounded by the length of S. This correspondence holds for infinite sequences and a sequence where $\mathtt{T} = \mathbb{N}$

Definition 2.4. *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 $Time^{\mathfrak{A}} = \{0, 1\}$

An application of progression: interactive simulation By iteratively applying progression inference, systems can be simulated with or without interaction.

Algorithm:

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.

3 Chapter 5

3.1 LTL

3.1.1 Syntax of LTL

Definition 3.1. The syntax of LTL-sentences (over Σ) is defined by the following BNF (Backus Naur form):

$$\varphi ::= \begin{cases} \bot \mid \top \mid p \text{ (where } p \in \Sigma) \mid \\ (\neg\varphi) \mid (\varphi \vee \varphi) \mid (\varphi \wedge \varphi) \mid (\varphi \Rightarrow \varphi) \mid \\ (\mathrm{X}\varphi) \mid (\mathrm{F}\varphi) \mid (\mathrm{G}\varphi) \mid (\varphi \mathrm{U} \varphi) \mid (\varphi \mathrm{W} \varphi) \mid (\varphi \mathrm{R} \varphi) \end{cases}$$

Informal semantics of LTL connectives An LTL formula is evaluated in a path of a transition structure $\mathcal{M}$.

- $\bot$: 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;
- $(\psi \text{ U } \phi)$: 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 expression is satisfied in a path in which ϕ is true in the first state.
- $(\psi \text{ W } \phi)$: *Weak-Until*: the same as U, except that it is satisfied as well if ψ remains true forever.
- $(\psi \text{ R } \phi)$: ψ *Releases* ϕ: ϕ is true at least until in a current or future state in which ψ is true.

Binding conventions of LTL

- Unary connectives bind most tightly
- Then in order: U,R,W,$\wedge, \vee, \Rightarrow$.

3.1.2 Formal semantics of LTL

Definition 3.2. Given $\mathcal{M} = \langle S, \rightarrow, L\rangle$ over Σ. Let $\pi = s_0 \rightarrow \ldots$ be a path in $\mathcal{M}$ and φ and LTL formula over Σ.

- $\mathcal{M}, \pi \models \top$ (and $\pi \not\models \top$)
- $\mathcal{M}, \pi \models p$ 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 $j = 0, \ldots, i-1, \mathcal{M}, \pi^j \models \psi$
- $\mathcal{M}, \pi \models \psi \ \mathrm{W} \ \varphi$ if
  - either there exists $i \geq 0$ such that $\mathcal{M}, \pi^i \models \varphi$ and for all $j = 0, \ldots, i-1, \mathcal{M}, \pi^j \models \psi$
  - or for all $j \geq 0, \mathcal{M}, \pi^j \models \psi$
- $\mathcal{M}, \pi \models \psi \ \mathrm{R} \ \varphi$ if
  - either there exists $i \geq 0$ such that $\mathcal{M}, \pi^i \models \psi$ and for all $j = 0, \ldots, i, \mathcal{M}, \pi^j \models \varphi$
  - or for all $j \geq 0, \mathcal{M}, \pi^j \models \varphi$

Definition 3.3. $\mathcal{M}, s \models \varphi$ if for each path $\pi = s \to \ldots$ in $\mathcal{M}$ starting in s, it holds that $\mathcal{M}, \pi \models \varphi$.

3.1.3 Other stuff

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.

CTL is required to express this sort of properties.

Definition 3.4. 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 $\mathcal{M}, \pi \models \psi$.

Duality All connectives and operators have a dual connective or operator. E.g., $\wedge$ and $\vee$ are called dual, since $\neg(\psi \wedge \varphi) \equiv \neg\psi \vee \neg\varphi$, and $\neg(\psi \vee \varphi) \equiv \neg\psi \wedge \neg\varphi$. Similarly, temporal operators have a dual operator:

- $\neg\mathrm{X}\phi \equiv \mathrm{X}\neg\phi$: X is self-dual
- $\neg\mathrm{G}\phi \equiv \mathrm{F}\neg\phi$, $\neg\mathrm{F}\phi \equiv \mathrm{G}\neg\phi$
- $\neg(\phi\mathrm{U}\psi) \equiv \neg\phi\mathrm{R}\neg\psi$, $\neg(\phi\mathrm{R}\psi) \equiv \neg\phi\mathrm{U}\neg\psi$

Other useful equivalences are as follows:

- $\mathrm{F}\phi \equiv \top\mathrm{U}\phi$, $\mathrm{G}\phi \equiv \bot\mathrm{R}\phi$, $\mathrm{G}\phi \equiv \phi\mathrm{W}\bot$

- $\phi \mathrm{U} \psi \equiv \phi \mathrm{W} \psi \wedge \mathrm{F} \psi$
- $\phi \mathrm{W} \psi \equiv \phi \mathrm{U} \psi \vee \mathrm{G} \phi$
- $\phi \mathrm{W} \psi \equiv \psi \mathrm{R}(\psi \vee \phi)$
- $\phi \mathrm{R} \psi \equiv \psi \mathrm{W}(\phi \wedge \psi)$

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.

Adequate sets of LTL connectives Each of the following sets is called an *adequate set of LTL connectives*:

$$\{\mathrm{U}, \mathrm{X}\}, \{\mathrm{R}, \mathrm{X}\}, \{\mathrm{W}, \mathrm{X}\}$$

3.2 Fairness constraints

Definition 3.5. A path π is fair with respect to some proposition φ if there are infinitely many $i \in \mathbb{N}$ such that $\pi^i \models \varphi$.

3.3 CTL*

3.3.1 Syntax of CTL*

Definition 3.6. We define *state formulas* and *path formulas* over vocabulary Σ by mutual induction using the following BNF:

- State formulas φ

$$\varphi ::= \begin{cases} \bot \mid \top \mid p \text{ (where } p \in \Sigma) \mid \\ (\neg\varphi) \mid (\varphi \vee \varphi) \mid (\varphi \wedge \varphi) \mid (\varphi \Rightarrow \varphi) \mid \\ \mathrm{A}[\alpha] \mid \mathrm{E}[\alpha] \end{cases}$$

- Path formulas α

$$\alpha ::= \begin{cases} \varphi \\ (\neg\alpha) \mid (\alpha \vee \alpha) \mid (\alpha \wedge \alpha) \mid (\alpha \Rightarrow \alpha) \mid \\ (\mathrm{X}\alpha) \mid (\mathrm{F}\alpha) \mid (\mathrm{G}\alpha) \mid \\ (\alpha \mathrm{U} \alpha) \mid (\alpha \mathrm{W} \alpha) \mid (\alpha \mathrm{R} \alpha) \end{cases}$$

Binding priorities for CTL*

- First unary connectives: $\neg$,X,F,G,A,E;
- Then, in order: U,R,W,$\wedge, \vee, \Rightarrow$

3.3.2 Semantics of CTL*

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 $\bot, \top, \text{atoms}, \neg, \wedge, \vee, \Rightarrow$;
  - $\mathcal{M}, s \models \mathrm{A}[\alpha]$ if for each path $\pi = s \to \dots$ in $\mathcal{M}$, it holds that $\mathcal{M}, \pi \models \alpha$
  - $\mathcal{M}, s \models \mathrm{E}[\alpha]$ if for some path $\pi = s \to \dots$ 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 X,F,G,U,W,R
  - if φ is a state formula: $\mathcal{M}, \pi \models \varphi$ if $\mathcal{M}, s_0 \models \varphi$

Definition 3.7. 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:

- $\mathrm{A}[\alpha] \equiv \neg\mathrm{E}[\neg\alpha]$
- $\mathrm{F}[\alpha] \equiv \neg\mathrm{G}\neg\alpha$
- $\alpha\mathrm{R}\beta \equiv \neg(\neg\alpha\mathrm{U}\neg\beta)$
- $\alpha\mathrm{W}\beta \equiv \alpha\mathrm{U}\beta \vee \mathrm{G}\alpha$

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. When using LTL formulas as path formulas, the universal path quantifier is used implicitly. In CTL*, this quantifier is to be explicated. See Definition 3.3 if unclear.

3.4 CTL

3.4.1 Syntax of CTL

CTL-formula's are CTL* state formulas in which all temporal connectives come in pairs:

Definition 3.8. The syntax of CTL-sentences (over Σ) is defined by the following BNF (Backus Naur form):

$$\varphi ::= \begin{cases} \bot \mid \top \mid p \text{ (where } p \in \Sigma) \mid \\ (\neg\varphi) \mid (\varphi \vee \varphi) \mid (\varphi \wedge \varphi) \mid (\varphi \Rightarrow \varphi) \mid \\ (\mathrm{AX}\varphi) \mid (\mathrm{EX}\varphi) \mid (\mathrm{AF}\varphi) \mid (\mathrm{EF}\varphi) \mid (\mathrm{AG}\varphi) \mid (\mathrm{EG}\varphi) \mid \\ (\mathrm{A}[\varphi\mathrm{U}\varphi)]) \mid (\mathrm{E}[\varphi\mathrm{U}\varphi)]) \end{cases}$$

Each CTL-formula is a CTL*-*state formula.*

3.4.2 Informal semantics of CTL

Expressions are evaluated in some node s of the tree:

- AXφ: for each path from s, φ holds in the next state;
  i.e., in all next states, φ holds
- EXφ: in some path from s, φ holds in the next state;
  i.e., in some next state, φ holds
- AFφ: each path goes through a state in which φ is true;
- EFφ: 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
- AGφ: in each path, φ is true in all states;
  i.e., φ is true in all states equal to or reachable from s
- EGφ: in some path, φ is true in all states;
- A[αUφ]: in each path, αUφ is true;
- E[αUφ]: in some path, αUφ is true;

3.4.3 Formulas expressible in 1, but not the other (LTL and CTL)

- In its normal state, a process can always request to enter its critical section:
  (use n_i: process i is in *normal* state, t_i: process i is applying for its critical section, c_i)
  - CTL: AG($n_1 \Rightarrow$ EXt_1)
  - LTL: not expressible in LTL
- If the process is enabled infinitely often, then it runs infinitely often: (use *enabled*, *running*)
  - LTL: GF *enabled* $\Rightarrow$ GF *running*
  - CTL: not expressible in CTL

Definition 3.9. We call $\mathcal{M}'$ a substructure of $\mathcal{M}$ if it is obtained by reducing edges and/or nodes from $\mathcal{M}$.

Theorem 2. *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 $\mathcal{M}', s \models \varphi$. Then φ is not expressible as an LTL formula.*

To show that AG EF p is not expressible in LTL, it suffices to find a structure in which it is satisfied but not in a substructure. See course notes for proof (currently page 161).

- A formula $\psi_1 \in CTL \setminus LTL$: AG EF p.
- $\psi_2 \in CTL \cap LTL$:

  - CTL: AG $(p \Rightarrow AFq)$
  - LTL: G $(p \Rightarrow Fq)$

- $\psi_3 \in LTL \setminus CTL$: G F$r \Rightarrow$ Fa (proof omitted in course notes). It means that in all paths in which r is infinitely often true, then a will be satisfied. This is a useful form of fairness: infinitely often requested (r) implies eventually acknowledged (a).
- $\psi_4 \in CTL^* \setminus (LTL \cup CTL)$: E[GF$p$] (proof omitted in the course notes). This means that there is a path with infinitely many p's.

3.4.4 Other stuff

Useful equivalences of CTL

- AX $\psi \equiv \neg$EX$\neg\psi$
- EG $\psi \equiv \neg$AF$\neg\psi$
- AG $\psi \equiv \neg$EF$\neg\psi$
- EF $\psi \equiv$ E[$\top$Uψ]
- A[ψUφ] $\equiv \neg$(E[$\neg\varphi$U$(\neg\psi \wedge \neg\varphi)$] $\vee$ EG$\neg\varphi$)

Adequate sets of CTL connectives A set of temporal connectives in CTL is adequate (i.e., suffices to express all CTL formulas) iff it contains one of {AX, EX} , at least one of {EG, AF, AU} and EU.

4 Chapter 6

Definition 4.1. 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.

Definition 4.2. 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 associated 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).

Note that this is the same idea as the verification method for proving invariants from bistate LTC theories.

Definition 4.3. 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:

$$\forall x, s : A \wedge I(s) \wedge J(s) \wedge H_e(x, s) \Rightarrow G_e(x, s)$$

- 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 4.4. 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:

$$\forall x, s, s' : A \wedge I(s) \wedge J(s) \wedge H_e(x, s) \wedge T_e(x, s, s') \Rightarrow Q_a(x, s, s')$$

- 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 ;
- T_e is the bistate formula expressing the effects of the concrete event refining e;
- Q_a is the bistate formula expressing the effects of the abstract action a.

T_e 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.

Correctness properties

Definition 4.5. 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.

Definition 4.6. 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.

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 Chapter 7

5.1 Deductive inference

Definition 5.1. A proof system consists of a set of *logical axioms* and *inference rules.*

Definition 5.2. An axiom schemata is an abstract formula containing formula variables. It represents the set of all formulas that can be obtained by substituting formula variables by formulas. E.g. $\alpha \vee \neg\alpha$, an instance of this schemata would be $P \vee \neg P$.

The inference rule $\frac{\beta_1,\ldots,\beta_n}{\alpha}$ specify that formula α can be inferred from formulas $\beta_1, \ldots, \beta_n$.

Definition 5.3. Given is a proof system:

- A formula φ is *provable* or *derivable* 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 $T \vdash \neg\varphi$

Definition 5.4. A proof theory is *sound* if $T \vdash \alpha$ implies $T \models \alpha$.

Definition 5.5. 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. Proof omitted (See course notes).

Theorem 3. Gödels Completeness theorem: *For each theory T and for each sentence α in FO, if $T \models \alpha$ then $T \vdash \alpha$*

5.1.1 Compactness theorem of FO

Theorem 4. Compactness Theorem of FO: *An infinite FO theory Ψ is satisfiable iff each finite subset is satisfiable.*

Proof. ($\Rightarrow$) If Ψ is satisfiable, it has a model which is also a model of all its finite subsets. Hence, each finite subset is satisfiable.

($\Leftarrow$) Assume towards contradiction that Ψ is not satisfiable[1]. By Gödels Completeness theorem, Ψ is inconsistent. By definition then, there exists a formula φ and a proof $Pr1$ of φ from Ψ and a proof $Pr2$ of $\neg\varphi$ from Ψ.
Let $\Omega = (Pr1 \cup Pr2) \cap \Psi$, the set of formulas of Ψ that are introduced in these proofs. Proofs are finite sequences; hence Ω is a finite subset of Ψ. By construction of Ω $Pr1$ and $Pr2$ are proofs from Ω, proving respectively φ and $\neg\varphi$. Hence, Ω is inconsistent. By soundness of $\vdash$, Ω is unsatisfiable. □

Equivalently, Ψ is unsatisfiable iff at least one finite subset is unsatisfiable. This can be used to prove inexpressivity in FO of propositions.

5.2 Some expressivity limitations of FO

Basic terminology and notations For logic formula φ, $Mod(\varphi)$ denotes the class of models of φ. For theory T, $Mod(T)$ denotes the class of models of T.

We say that a class $\mathcal{C}$ 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 $\mathfrak{A}' \in \mathcal{C}$.

Expressing classes of structures

Definition 5.6.
- 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 $\mathcal{L}$.

[1]The proof in this direction uses contraposition. The way the beginning and end of this direction is phrased in the course notes is misleading.

5.2.1 Inexpressivity theorem

Theorem 5. *Let $\mathcal{C}$ be a class of structures and let T' be an infinite theory such that:*

- *T' is inconsistent with $\mathcal{C}$;*
  or formally, $\mathcal{C} \cap Mod(T') = \emptyset$;
- *each finite subset of T' is consistent with $\mathcal{C}$;*
  or formally, $\mathcal{C} \cap Mod(\Omega) \neq \emptyset$, for each finite subset $\Omega \subseteq T'$.

Then $\mathcal{C}$ is not expressible 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 $\Omega' = \Omega \cap T'$. 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
$\square$

Finiteness of the universe cannot be expressed

Theorem 6. "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} (See page 192 for this theory. It is the theory with an infinite amount of propositions where proposition i states that there are at least i different elements in the universe.). 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 atleast 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.
$\square$

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.

Reachability cannot be expressed in FO The proposition "There is no path from A to B in graph G" can be infinitely expressed. See page 195 in the course notes for definition of the theory. Proposition i in this theory says that there is no path of length i between A and B in graph G. This theory is denoted as $T_{ABUncon}$.

Theorem 7. "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 $\mathcal{C}_{ABCon}$. This class is inconsistent with $T_{ABUncon}$ but consistent with each finite subset Ω of $T_{ABUncon}$. 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 $\mathcal{C}_{ABCon}$. The inexpressivity theorem applies. □

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 τ . This proposition is denoted as DCA(τ).

Recall that the universe (also called domain) of a structure $\mathfrak{A}$ is denoted as $D_{\mathfrak{A}}$. A structure $\mathfrak{A}$ satisfies DCA(τ) iff

$$D_{\mathfrak{A}} = \{t^{\mathfrak{A}} \mid t \in S_\tau\}$$

Hence, the class $\mathcal{C}_{DCA(\tau)}$ characterized by DCA(τ):

$$\mathcal{C}_{DCA(\tau)} = \{\mathfrak{A} \mid D_{\mathfrak{A}} = \{t^{\mathfrak{A}} \mid t \in S_\tau\}\}$$

Theorem 8. *If τ contains at least one constant and one function symbol, then DCA(τ) is not expressible in FO.*

Proof. 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 DCA(τ). Hence, $\mathcal{C}_{\mathcal{DCA}(\tau)}$ is inconsistent with T_a.

However, $\mathcal{C}_{\mathcal{DCA}(\tau)}$ 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 $\neg(a = t') \notin \Omega$, 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. □

5.3 Undecidability and Gödels incompleteness theorem

Theorem 9. (Undecidability of FO). *The validity problem for FO is undecidable. That is, the set $\{\alpha \mid \ \models \alpha\}$ is undecidable.*

There is no terminating algorithm that for input α answers **T** if $\models \alpha$ and **f** otherwise.

Proof. Proof omitted in the course notes. □

Theorem 10. (Semi-decidability of FO). *The validity problem for first order logic is semi-decidable. That is, the set $\{\alpha \mid \ \models \alpha\}$ is semi-decidable.*

I.e., there is an algorithm that for input α answers **t** if $\models \alpha$ and otherwise returns **f** or does not terminate.

Take note that these two theorems are not mutually exclusive.

Proof. We do not give the proof but the basic idea is simple. Take the following algorithm for deciding $\models \alpha$: generate all proofs of some sound and complete proof system and return t if a proof for α is found. If α is valid, a proof exists and will be found and the algorithm will terminated with **t**. If α is not valid, the algorithm will not terminate. This proves the recursive enumerability of $\{\alpha \mid \ \models \alpha\}$. □

Gödels incompleteness theorem

Definition 5.7. Let τ be a finite set of symbols.

- A string set S over τ is *recursive* or *decidable* if there is an algorithm that for given input string x terminates and answers **t** if $x \in S$ and answers **f** otherwise.
- A string set S over τ is *recursively enumerable* if there is an algorithm that outputs every element of S (potentially in infinite time).
- A string set S over τ is *semi-decidable* if there is an algorithm that for given input string x terminates and answers **t** if $x \in S$ and answers **f** or does not terminate if $x \notin S$.

Theorem 11. (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 $T \not\models \alpha$*

Corollary 11.1. *If T is satisfied by $\mathbb{N}$ and entails all formulas φ true in $\mathbb{N}$, then T is not recursively enumerable. In particular, theory($\mathbb{N}$) is not recursively enumerable.*

Theorem 12. *A string set S over τ is recursively enumerable if and only if S is semi-decidable.*

Proof. Proof omitted. □

Corollary 12.1. (undecidability of the natural numbers). *The problem of deciding truth of an FO sentence in $\mathbb{N}$ is undecidable and not semi-decidable.*

This is a direct consequence of Gödels incompleteness theorem.

Corollary 12.2. *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.*

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.

6 Chapter 8

6.1 Overview of inference problems

Illustrated with examples from the course notes.

1. Propagation inference:

   - Input: T_c, partial structure $\mathcal{I}$
   - 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 $\mathcal{I}' \leq_p \mathfrak{A}$.

   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 $\mathcal{I}'$.

   This is applied every time the user selects a new course. The old structure is expanded with the new choice and the system shows what new restrictions have appeared due to the consequences of that choice.

2. Model checking inference:

   - Input: $T_c, \mathfrak{A}$
   - Output: whether or not $\mathfrak{A} \models T_c$

The user can make the system check the current structure and see if it is a valid planning of courses.

3. Explanation inference: An explanation is a logical argument why an axiom or group of axioms is violated, why a certain choice is forced or impossible.

   Applying this, the user can see why some planning is not valid.

4. Model expansion inference:

   - Input: $T_c, \mathcal{I}$
   - Output: a model $\mathfrak{A}$ of T_c such that $\mathcal{I} \leq_p \mathfrak{A}$.

   The user can let the system decide on courses if they do not have a preference.

5. Optimisation inference:

   - Input: $T_c, \mathcal{I}, t$ 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.

   The user can use this to compute the model expansion with the minimal study load.

6. Query answering:

   - Input: $\mathfrak{A}, \varphi$ or $\{x : \varphi[x]\}$
   - Output: $\mathfrak{A} \models \varphi$ or $\{x : \varphi[x]\}^{\mathfrak{A}}$

7. Δ-model expansion:

   - Input: a $Param(\Delta)$-structure $\mathfrak{A}_p$
   - Output: the structure $\mathfrak{A}$ satisfying Δ expanding $\mathfrak{A}_p$

8. 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 $\mathfrak{A}$.

Knowledge base paradigm This paradigm separates knowledge representation (information) from the inference tasks (problems) on it. Currently, these two aspects are commonly linked to each other (FO - deductive inference, SQL - query inference, ...). In IDP, these are separate. One only needs to update the main theory to enable all sorts of inferences to be able to reason about the theory. If these were linked, one would need to update the different theories linked with the different inferences. This enables high reuse of the theory. Problems that require multiple different forms of inference working together can be solved like this.

7 Chapter 9

7.1 Proving satisfiability of Propositional Logic

Definition 7.1.

- A *literal* is an atom p or the negation of an atom $\neg p$.
- A formula is in *negation normal form* (NNF) if it only contains $\wedge, \vee, \neg$ and the negation symbol $\neg$ only occurs in literals.
- A *clause* is a disjunction of literals.
- A formula is in *conjunction normal form* (CNF) if it is a conjunction of clauses.

Theorem 13. *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. □

Theorem 14. *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, $\varphi_1 \wedge \cdots \wedge \varphi_n$ 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. □

Theorem 15. *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 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. $(\psi \Leftrightarrow \phi) \Leftrightarrow (\psi \Rightarrow \phi) \wedge (\phi \Rightarrow \psi)$
2. $(\psi \Rightarrow \phi) \Leftrightarrow (\neg\psi \vee \phi)$
3. $\neg\neg\psi \Leftrightarrow \psi$
   $\neg(\psi \wedge \phi) \Leftrightarrow (\neg\psi \vee \neg\phi)$
   $\neg(\psi \vee \phi) \Leftrightarrow (\neg\psi \wedge \neg\phi)$
   After this step, we obtain formulas in NNF.
4. $(\psi \wedge \phi) \vee \delta \Leftrightarrow (\psi \vee \delta) \wedge (\phi \vee \delta)$

We denote the result of applying this algorithm on φ as $ToCNF(\varphi)$. Step 1 and 4 create two copies of subformulas. Hence, they may double the size of the formula. In the worst case, the formula is blown up exponentially.

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:

$$\{p_\varphi, ToCNF(p_\psi \Leftrightarrow \psi') \mid \psi \text{ is } \varphi \text{ or a non-literal subformula of } \varphi\}$$

Theorem 16. *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. □

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.

7.2 SAT algorithms

7.2.1 DPLL: a basic SAT solver

Definition 7.2. 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"). 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.

Algorithm 1 Unit Propagation

```
1: procedure UP($\mathcal{I}$)
2:     while there exists a unit clause with unit literal $L$ do
3:         $\mathfrak{A}(L) := \mathbf{t}$
4:         if there exists a conflict clause then
5:             return "Conflict"
6:     return
```

The DPLL algorithm A high level description of DPLL is given. 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: $\mathcal{I}(L) := \mathbf{f}$.

Algorithm 2 DPLL

```
1: procedure DPLL($\varphi$)
2:     $\mathcal{I}(P) := \mathbf{u}$, for all P in $\varphi$
3:     while true do
4:         UP
5:         if "Conflict" then
6:             Backtrack
7:         else if $\mathcal{I}$ is a model then
8:             return $\mathcal{I}$
9:         else
10:            Decide
11:    return
```

Analysis of DPLL At any point of time, the state of DPLL can be represented as a sequence of labeled literals:

$$\langle L_0^{\kappa_0} L_1^{\kappa_1} \dots L_n^{\kappa_n} \rangle$$

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.

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.

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.

7.2.2 Conflict-Driven Clause learning (CDCL)

See course notes for illustrative examples coupled with the explanation.

Definition 7.3. The resolution of two clauses $\psi = p \vee L_1 \vee \dots \vee L_n$ and $\phi = \neg p \vee L'_1 \vee \dots \vee L'_m$ on p is the *resolvent* $\delta = L_1 \vee \dots \vee L_n \vee L'_1 \vee \dots \vee L'_m$.

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.

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.

First resolution step Assume that during execution, a conflict arises due to the conflict clause $L_1 \vee \dots \vee L_n$. 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.

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, $\neg L_1$ 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**.

Algorithm 3 Clause Learning

```
1: procedure ClauseLearning(ConflictClause)
2:     while ConflictClause contains $\geq 2$ literals of current level do
3:         ConflictClause = ResolveYoungest(ConflictClaus)
4:     return ConflictClause
```

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.

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):

$$L_1 \vee L_2 \cdots \vee L_k (k \geq 1)$$

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 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.

Algorithm 4 Backjumping

```
1: procedure BACKJUMP(LearnedClause)
2:     Determine $m$ from LearnedClause
3:     Undo all assignments to literals of level $m + 1, m + 2, \ldots, n$
```

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. See course notes for an illustrative example.

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($\mathfrak{A}$).

Algorithm 5 Conflict-driven clause learning

```
procedure CDCL($\varphi$)                    ▷ prints all models; UNSAT if list is empty
    Initialize $\mathcal{I}(P) \leftarrow \mathbf{u}$, for all $P \in \Sigma$
    while true do
        UP
        if $\mathcal{I}$ is a model then
            Print $\mathcal{I}$
            ConflictClause := NoModel($\mathfrak{A}$)
            Conflict := true
            Add ConflictClause to Clause database
        if conflict and current level is 0 then
            return "Unsat"
        if conflict and current level > 0 then
            LearnedClause := ClauseLearning(ConflictClause)
            Backjump(LearnedClause)
        else                          ▷ no conflict and there are undefined symbols
            Decide
```

7.2.3 Optimisations

See course notes for optimisations

8 Chapter 10

8.1 CTL-model checking algorithm

In this section, we see an algorithm for solving 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 solve 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 $\{\mathrm{EG}, \mathrm{EU}, \mathrm{EX}, \wedge, \neg, \bot\}$.

Proposition 16.1.

- $\mathcal{M}, s \models p \Leftrightarrow p \in L(s)$
- $\mathcal{M}, s \models \psi_1 \wedge \psi_2 \Leftrightarrow \mathcal{M}, s \models \psi_1 \wedge \mathcal{M}, s \models \psi_2$
- $\mathcal{M}, s \models \neg\psi_1 \Leftrightarrow \mathcal{M}, s \not\models \psi_1$
- $\mathcal{M}, s \models \mathrm{EX}\psi_1$ if there exists $s \to s'$ such that $\mathcal{M}, s' \models \psi_1$
- $\mathcal{M}, s \models \mathrm{E}(\psi_1 \mathrm{U} \psi_2)$ iff there exists a finite path $s = s_0 \to s_1 \to \cdots \to s_n (n \geq 0)$ 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 $O(1)$.
- For each S_ψ, checking $s \in S_\psi$ and adding s to S_ψ is $O(1)$.

The algorithm computes S_ψ by a case analysis on ψ.

- $\psi = \bot : S_\psi = \emptyset$
- $\psi = p \in \Sigma : S_\psi = \{s \in S \mid p \in L(s)\}$;
  To compute S_ψ, 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_ψ 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 $s \in S_{\psi_1}$ and $s \in S_{\psi_2}$, 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 $O(V + E)$.

- $\psi = \mathrm{E}(\psi_1 \mathrm{U} \psi_2) : S_\psi = \{s \in S \mid \exists s_0, \ldots, s_n : s = s_0 \to s_1 \to \cdots \to s_n \wedge \forall i \in \{0, \ldots, n-1\} : s_i \in S_{\psi_1} \wedge s_n \in S_{\psi_2}\}$;
  The set S_ψ is computed by the following algorithm. It uses a queue datastructure Q for which pushing and popping an element is $O(1)$.

```
1: $S_\psi := Q := S_{\psi_2}$
2: while $Q \neq \emptyset$ do
3:     pop $s$ from $Q$
4:     for every $s' \to s$, if $s' \in S_{\psi_1}$ add $s'$ to $S_\psi$ and to $Q$.
5: return $S_\phi$
```

  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.

```
1: $S_\psi := S_{\psi_1}$
2: for $s \in S_\psi$ do
3:     if $s$ has no edge $s \to s'$ such that $s' \in S_\psi$ then
4:         delete $s$ from $S_\psi$; restart for loop
5: return $S_\phi$
```

  Every time an element of S_ψ 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 $O(V \times (V + E))$.

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 M. This is too expensive. Indeed, the size of M tends to be very large.

A more efficient computation of $S_{\mathbf{EG}\psi_1}$ 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;

$$S' = S_{\psi_1} \text{ and } \to' = \{(s, s') \in S'^2 \mid s \to s'\}$$

- Compute all maximal strongly connected components (SCC's) of $\to'$. A strongly connected component of a graph is a set of states such that each state in it is reachable from each other state through that. 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 $O(V + E)$.

In conclusion, the complexity of the refined algorithm is $O(f \times (V + E))$ and is linear in the size of $\mathcal{M}$.

8.1.1 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. In that case, the specification of the dynamic system is a pair $\langle \mathcal{M}, \mathcal{C} \rangle$ of a transition structure $\mathcal{M}$ and a set C of fairness constraints. Paths of $\langle \mathcal{M}, \mathcal{C} \rangle$ are paths of $\mathcal{M}$ that satisfy all fairness constraints $c \in C$.

Definition 8.1. 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 8.2. The paths of a system $\langle \mathcal{M}, \mathcal{C} \rangle$ 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 $c \in C$.