Summary Knowledge Representation

Phillip Lippe

March 28, 2019

Contents

1 Introduction to KR

- There are two main lines of development in AI: *symbolic* and *statistical* representation
- Both approaches go along with different benefits/weaknesses

	Symbolic	Connectionist
Construction	Human effort	Data hunger
Scaleable	+/-	+/-
Explainable	+	-
Generalisable	Performance cliff	Performance cliff

Figure 1: Weaknesses of symbolic and statistical representation in AI

  - ***Construction***: effort that is needed to create such a system
    * A symbolic AI requires a knowledge base on which it bases its reasoning. This knowledge base is mostly created by a human which can take a lot of time. For example, the SNOMED database was created in more than 40 years and contains now about $300,000$ definitions. The construction of knowledge bases is summarized in the research area *Knowledge Engineering*.
    * In the connectionist/statistical approach, the model learns from data so that we need to provide a (huge) dataset. Depending on the task and the required labels, it can also take a lot of human effort until we have the required data size.
  - ***Scalability***: effect of data size/amount on the systems
    * The more data we have for an symbolic AI, the easier it is to run into a problem. Huge knowledge bases tend to be not sound anymore (consistent) as a small mistake at one point can lead to wrong reasoning for any problem (if knowledge base is unsatisfiable, then all given problems are unsatisfiable), So we need to put extra effort in ensuring the soundness of the knowledge base.
    * Connectionist approaches learn from the statistics of a dataset which gets more accurate by increasing the amount of data. Small errors/noise are thereby smoothed out. However, this also means that statistical representations are inaccurate if there is only a small amount of data.
  - ***Explainable***: understanding how the system came to its decision
    * Symbolic AI is dedicated to creating explainable systems as they only applies facts/statements/rules of the knowledge base that was hand-crafted and thus understandable for a human. The reasoning of such an AI system is explainable by the used and newly derived rules.
    * In contrast, connectionist approaches are less explainable. As they capture the data distribution in a very high dimensional space (e.g. neural networks with millions of parameters), it more serves as a black box. Errors that are produced by small, carefully selected input noises are harder to understand and to prevent.
  - ***Generalization***: performance across unseen domains
    * Symbolic AI relies on the given knowledge base. If we try to reason about a domain that is unknown in the knowledge base, we don't get any answer (or rather that the reasoning ended without a result). For example, if we have a system based on SNOMED and try to show that if it is raining outside, I probably get wet the AI terminates without an solution because it is not provided by the needed rules/facts.
    * Commonly, statistical AI systems are already limited to their specific domain. In the task of classification, we usually have a fixed set of classes from which the system has to choose one. If we show a new image that does not belong to any of those, it will try to find the most similar class of what the system has so far (borders in high-dimensional space).

2 Satisfiability solvers

2.1 Propositional Logic

- In Knowledge Representation, we have three sets of rules/formula:
  - *Knowledge base*: statements which are known to be true/need to be fulfilled by all models (*axioms*)

  - *Premises*: statements that are only true for certain states/inputs (*implications* at partial truth value assignment)
  - *Conclusion*: statements for which we want to check if there exists an model for (*conjecture*). Derived by knowledge base and premises
- Propositional logic is based on simple statements (*literals*) that can be combined to complex statements (*formula*)
  - *Conjunction*: $A \wedge B$
  - *Disjunction*: $A \vee B$
  - *Negation*: $\neg A$
  - *Implication*: $A \rightarrow B$ ($\equiv \neg A \vee B \equiv \neg(A \wedge \neg B)$)
- Truth values are assigned to literals by an interpretation function I
- **Clausal normal form**
  - Every formula can be rewritten in CNF (conjunction of disjunctions)
  - Example: $(A \vee B \vee C \vee D) \wedge (E \vee F) \wedge (\neg A \vee F \vee D) \wedge ...$
  - To rewrite a formula to CNF, we have to remove implications ($A \rightarrow B \equiv \neg A \vee B$), move negations in front of the literals ($\neg(A \vee B) \equiv \neg A \wedge \neg B$) and move conjunctions outside ($A \vee (B \wedge C) \equiv (A \vee B) \wedge (A \vee C)$)
- Propositional logic is a weak language i.e. it is less expressive (no instances, no functions on terms, ...)
- We can express first-order logic in propositional logic by instantiating all quantifiers and all possible input arguments to predicates
  - Only possible for finite domains. Exponential explosion of number of instances
  - Example: Domain $\{A, B, C\}$, $\forall x P(x) \vee Q(x, x) \implies (P\_A \vee Q\_A\_A) \wedge (P\_B \vee Q\_B\_B) \wedge (P\_C \vee Q\_C\_C)$
  - $\exists x \forall y Q(x, y) \implies (Q\_A\_A \wedge Q\_A\_B \wedge Q\_A\_C) \vee (Q\_B\_A \wedge ...) \vee ...$
- *Tautology*: formula that is always true (e.g. $A \vee \neg A$), also called valid sentence. For n symbols, we have 2^n models.
- *Contradiction*: formula that is always false (e.g. $A \wedge \neg A$), also called inconsistent sentence
- *Satisfiable sentence*: formula that can be made true in at least one world (i.e. not inconsistent)
- A *model* is a "possible world" (truth assignment) in which the knowledge base is true (including conjecture if given)
- $P \models Q$ means that any model of P also fulfills Q

2.2 Davis Putman algorithm

- Algorithm for proving if there exists an model for a given knowledge base, **and** returns it as a result in that case
- In general, this problem is NP-complete
  - Any other NP problem can be reduced to SAT
  - Exponential time $\mathcal{O}(2^n)$ with number of literals, but algorithms are optimized to be fast for most cases (only worst case is exponential)
- The DPLL algorithm is summarized in Figure 2 and described in more detail here:

Step 1 **Simplification**: iteratively remove unnecessary clauses and derive literal implications
  - *Tautology*: remove tautologies like $P \vee \neg P$ from knowledge base (once in the beginning)
  - *Pure literals*: set predicates that solely occur in their positive *or* negative form to the corresponding truth value
  - *Unit clauses*: set literals for which the knowledge base contain a unit clause to true (or the predicate to false respectively)

Step 2 **Split**: pick a predicate and assume a truth value

- Heuristics of which literal to pick next can improve the efficiency of DPLL a lot
- **DLCS** (*Dynamic Largest Combined Sum*): Pick v with the largest count of positive and negative occurrences: $CP(v) + CN(v)$. If $CP(v) > CN(v)$, choose $v = 1$, else $v = 0$
- **DLIS** (*Dynamic Largest Individual Sum*): Pick v with either largest CP or CN. Same truth assignment as for *DLCS*.
- **Jeroslow-Wang**: weight of literal depends on the length of clauses it occurs in. Thereby, we prefer small clauses. The score of a literal is $J(v) = \sum_{c \in \mathcal{C}_v} 2^{-|c|}$, and we pick the highest value. One-sided JW looks at v and $\neg v$ independently, whereas the two-sided approach looks at sum of v and $\neg v$, and just picks the truth value based on $J(v) \geq J(\neg v) \Rightarrow v = 1$, else $v = 0$.
- **MOMs** (*Maximum Occurrences in Clauses of Minimum Size*): similar to JW, but we only look at the smallest clauses in the knowledge base. The number of occurrences in those is indicates by the function f. We choose the literal that maximizes $[f(v) + f(\neg v)] * 2^k + f(v) \cdot f(\neg v)$. k is a tuning parameter for the trade-off between the balanced distribution of v and $\neg v$ and their individual ones.

Algorithm 1.1: `DPLL-recursive`(F, ρ)

```
Input  : A CNF formula F and an initially empty partial assignment ρ
Output : UNSAT, or an assignment satisfying F
begin
    (F, ρ) ← UnitPropagate(F, ρ)
    if F contains the empty clause then return UNSAT
    if F has no clauses left then
        Output ρ
        return SAT
    ℓ ← a literal not assigned by ρ                    // the branching step
    if DPLL-recursive(F|ℓ, ρ ∪ {ℓ}) = SAT then return SAT
    return DPLL-recursive(F|¬ℓ, ρ ∪ {¬ℓ})
end

sub UnitPropagate(F)
begin
    while F contains no empty clause but has a unit clause x do
        F ← F|x
        ρ ← ρ ∪ {x}
    return (F, ρ)
end
```

Figure 2: DPLL pseudo-code algorithm

2.2.1 Clause Learning

- *Non-chronological backtracking*: If we encounter a conflict at the end of a search branch, we try to find the root of the problem, and then backtrack to the that point.
- The implications (decisions/assignments) that lead to a conflict can be modeled as an acyclic graph where each node represents a literal assignment and each edge represents the reason for that assignment (see Figure 3)
- We can then partition the graph such that one side contains at least all decision variables (called *reason side*), and the other the conflict literal (called *conflict side*).
- The conflict clause is determined by the negations of the literals associated with the cut between both sides.
- Different cuts of the implication graph distinguish learning schemes from one another as they imply different conflict clauses and hence the information gained from them.
- Conflict clause learning is a selective application of resolution (probably more useful than random resolution). General resolution rule:

$$\frac{A \vee \neg B \qquad C \vee B}{A \vee C}$$

2.3 Stochastic solver

- Properties of a SAT solver
  - *Decidability* = completeness.

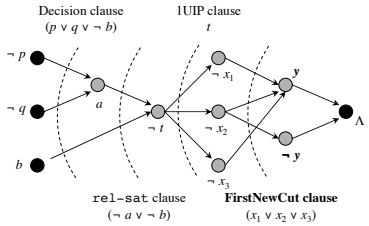


Figure 3: Conflict clause implication graph

  - ∗ This means that given enough runtime, the SAT solver guarantees to find an assignment or returns that there is no solution.
  - ∗ Although DPLL is complete in theory, it is not in practice as we have limited runtime to wait for an answer
  - ∗ Note that *undecidability* just indicates that it is not *always* guaranteed to get an answer (may be harmless in practice)
- – *Complexity*: exponential maximum runtime $\mathcal{O}(n^2)$
  - ∗ However, mean instead of worst-case runtime is more important in practice
  - ∗ Also, $\mathcal{O}$-notation does not consider constants whereas with limited runtime, it might be important

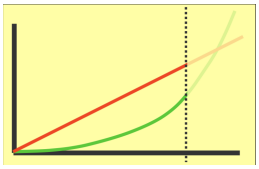
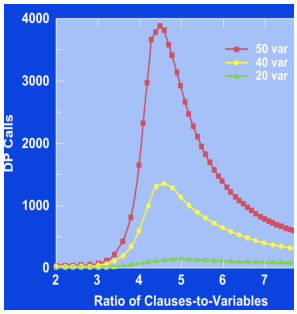
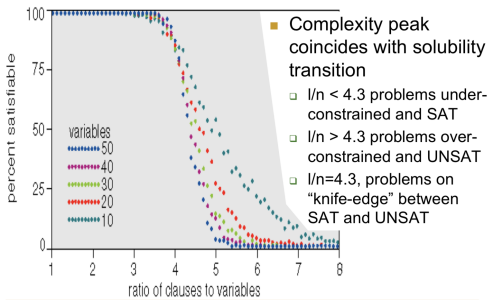


Figure 4: Comparison of worst case runtime $\mathcal{O}(n)$ and $\mathcal{O}(n^2)$ with different constants

- A good measurement for complexity in case of SAT solvers has been shown to be the ratio of clauses to variables (see Figure 5)
- Problems with a low ratio are easy to solve as they have (in average) many solutions
- Problems with a high ratio tend to have no solution, and are easy to show that there exists a conflict (see Figure 5b)
- In between, around 4.26, are apparently the hardest randomly generated problems as there might a solution (or only a few) or not. This points is also referred to as *phase transition*



(a) Problem complexity



(b) Proportion of satisfiable problems

Figure 5: Hardness in SAT problems

- Stochastic solvers have been shown to perform quite well on such randomly generated SAT instances, but might perform poorly compared to DPLL on highly structured problems
- It is not intended to find all solution neither that there exists none. But is for example useful for MAXSAT (see later)
- Stochastic SAT solvers perform local search

1. Make a guess (smart or random) about values of the variables
2. Evaluate how many clauses are broken
3. Try flipping a variable to make things better for a certain number of iterations (various heuristics in which variables to flip next)

- This search is repeated N times until we either find a solution or terminate without an answer
- **GSAT**: Local greedy search/algorithm of picking the next variable to flip which increases the number of satisfied clauses the most (ties are broken randomly, note that flips will break also some clauses).
- Full algorithm shown in Figure 6
- Note that this algorithm tends to get stuck in local minimum (flipping a single variable does not increase score). Thus, we perform random restarts to start new. That's also why GSAT spends most time on plateaus where score is not improved

```
procedure GSAT(Sigma)
  for i := 1 to MAX-TRIES ; These are the restarts
    T := random(Sigma) ; random assignment
    for j := 1 to MAX-FLIPS ; To ensure termination
      if T satisfies Sigma then return T
      else T := T with variable flipped to maximize
                number of satisfied clauses
      ; It doesn't matter if the number does
      ; not increase.  This are the sideways flips
    end
  end
```

Figure 6: Pseudo-code of GSAT

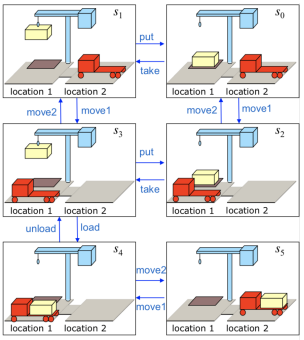
2.4 MAXSAT

- MAXSAT is Proportion *optimization* extension of SAT that asks what is the maximum number of clauses that can be simultaneously satisfied
- Example: $\neg A \wedge (A \vee B) \wedge (\neg B)$ is a contradiction. But the truth assignment $\pi = \{\neg A, \neg B\}$ maximizes number of satisfied clauses
- Some clauses might be more important to be satisfied than others $\Rightarrow$ adding a cost/positive weight to every clause C that will be incurred if C is falsified
- A cost of ∞ indicates *hard* clauses that are mandatory to satisfy, whereas *soft* clauses have a finite cost
- We try to minimize the sum of the costs of all unsatisfied clauses
- There are different variations of MAXSAT:
  - (Standard) MAXSAT: no hard clauses and all have a weight of 1 (solution maximizes the number of satisfied clauses)
  - *Weighted* MAXSAT: no hard clauses, but soft clauses with any finite, positive weight
  - *Partial* MAXSAT: hard clauses are allowed, but all soft clauses have weight 1
  - *Weighted Partial* MAXSAT: both hard and soft clauses are allowed. Subsumes all previously mentioned variations.

2.5 SAT planning

- Real-world planning problems can be translated to a SAT problem, and the plan can be extracted from the truth assignments

- Formal definition of the planning problem
  - States $S = \{..., s_i, ...\}$ (in the example $S = \{s_0, s_1, s_2, s_3, s_4, s_5\}$)
  - Actions $A = \{..., a_i, ...\}$ (in the example $A = \{\texttt{move1}, \texttt{move2}, \texttt{put}, \texttt{take}, \texttt{load}, \texttt{unload}\}$)
  - State-transition function $\gamma : S \times A \to S$ (note that we only consider deterministic transitions, otherwise 2^S output)
  - Planning problem $P = (\Sigma, s_0, s_G)$ where $\Sigma = (S, A, \gamma)$, and s_0 initial state and s_G (set of) goal states
  - Classical plan is a sequence of actions: $\pi = \langle a_0, a_1, ..., a_{n-1} \rangle$
  - Policy is a partial function from S to A



- Translation for plan P and a fixed plan length n:
  - If $\pi = \langle a_0, a_1, ..., a_{n-1} \rangle$ is a solution to the planning problem, we know that the traversed states are $s_0, s_1 = \gamma(s_0, a_0), s_2 = \gamma(s_1, a_1), ..., s_n = \gamma(s_{n-1}, a_{n-1})$ (where a_i is the i-th step of π, and s_i the states in which the agent is at step i)
  - We denote all possible literals with L
  - Formula describing the initial state: $\bigwedge \{l_0 \mid l \in s_0\} \wedge \bigwedge \{\neg l_0 \mid l \in L - s_0\}$
  - Formula describing the goal state: $\bigwedge \{l_n \mid l \in g^+\} \wedge \bigwedge \{\neg l_n \mid l \in g^-\}$ (g^+ valid goal states, g^- invalid goal states)
  - Formula describing the state-transitions. For every a_i add $\bigwedge \{p_i \mid p \in \text{Precond}(a)\} \wedge \bigwedge \{e_{i+1} \mid e \in \text{Effects}(a)\}$
  - Complete exclusion axiom: only one action per step ($\neg a_{1,i} \vee a_{2,i}$,...)
  - Frame axioms that describe what *doesn't* change between i and $i+1$: $\left(\neg l_i \wedge l_{i+1} \Rightarrow \bigvee_{a \in A} \left\{a_i | l \in \text{Effects}^+(a)\right\}\right) \wedge \left(l_i \wedge \neg l_{i+1} \Rightarrow \bigvee_{a \in A} \left\{a_i | l \in \text{Effects}^-(a)\right\}\right)$ ($\text{Effects}^+(a)$: set of literals that change their truth value to true if a is performed, and $\text{Effects}^-(a)$ those that are changed to false)
- We apply a SAT solver on this knowledge base. If we find a solution, then P has a solution of length n
- To find solutions of shortest length, we loop over different values for n. If we don't find any solution for $n = 0$, we encode the problem for $n = 1$, and so on.
- In practice, SAT solvers for planning take too much time and memory, but can be combined with other techniques like planning-graph expansion (SatPlan).

2.6 Applications of SAT

- Model checking in terms of hardware and software verification (can state S be ever reached? Is state T always reached after S?)
- Classical planning
- Combinatorial design (existence of mathematical structures)
- Solving subproblems in domains like scheduling, test pattern generation, multi-agent systems, ...

3 Description Logic

- Description Logic is the logic for ontologies
- Is more expressive than propositional logic, but less than first-order logic (DL is subsumed by FOL as classes are unary predicates and relations binary)
- We limit our discussion to the $\mathcal{ALC}$ description logic (*Attributive Concept Language with Complements*)

3.1 Ontologies

- Organizing knowledge in a way that is useful for people
- Fundamental elements of ontologies
  - *Class/Type/Concept*: name + set of properties that describe certain set of individuals
  - *Instances*: members of the set defined by the class
  - *Property/Relation*: assert facts about the instances/relation between classes
- The backbone of every ontology is a type-/class-hierarchy where multi-parent inheritance is possible
- Axioms that need to be formalized in a logic for ontologies:
  - Two classes are equivalent iff they have the same individuals and same definition
  - The intersection and union of classes
  - A class can be specified by either its definition or enumeration of all members
  - Restriction of some or all values from the specified class
- Property types/possible properties of relations:
  - *Symmetry*: if a to b by relation r, then b to a by r
  - *Asymmetry*: if a to b by relation r, then there can't be b to a by r
  - *Transitivity*: if a to b and b to c by relation r, then a to c by r
  - *Functionality*: if a to b and a to c by relation r, then b and c must be the same
  - *Inverse functionality*: if a to b and c to b by relation r, then a and c must be the same
  - *Reflexivity*: a to a by relation r always holds
  - *Ir-reflexivity*: a to a by relation r can never hold
  - *Inverse property*: if a to b by relation r, then b to a by relation q

3.2 Fundamentals of Description Logic

- A logic is defined by
  - A language ("syntax")
  - The meaning of expressions ("semantics")
  - Inference ("deduction")

3.2.1 Syntax

- The vocabulary of DL includes concept/class/type and role names
- Furthermore, we have the universal concept $\top$ (everything) and the bottom concept $\bot$ (nothing)
- More complex types can be set together by union $\sqcup$, intersection $\sqcap$ and complement $\neg$
- Restrictions are encoded by $\exists r.C$ and $\forall r.C$. See semantics for meaning/explanation, and visualization in Figure **??**

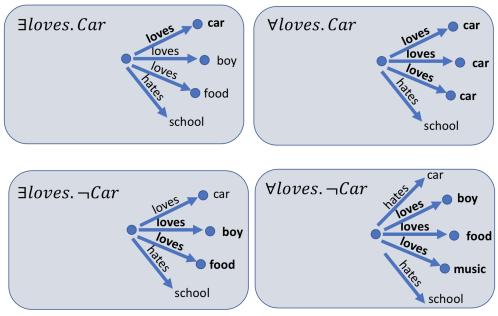


Figure 7: Visualization of restriction operators $\exists r.C$ and $\forall r.C$

- Examples:
  - $\texttt{Boys} \sqcup \texttt{Girls}$
  - $\texttt{Girls} \sqcap \exists \texttt{owns}.\texttt{Car}$
- Important for identifying concepts and roles in text. Example:

  "Any artwork is created by an artist. A sculpture is an artwork. A painting is an artwork that is not a sculpture. A painter is someone who painted a painting. A sculptor is someone who sculptured an artwork and only create sculptures. If an artwork is created by an artist, he has either painted or sculptured it."

  The solution would be the concepts $\{\texttt{Artwork}, \texttt{Artist}, \texttt{Sculptor}, \texttt{Painter}, \texttt{Painting}, \texttt{Sculpture}\}$, and the roles $\{\texttt{created}, \texttt{created\_by}, \texttt{painted}, \texttt{sculptured}\}$

  "*An artwork that is not a sculpture*": $\texttt{Artwork} \sqcap \neg \texttt{Sculpture}$

  "*Some who painted a painting*": $\exists \texttt{painted}.\texttt{Painting}$

  "*Someone who sculptured an artwork and only created sculptures*": $\exists \texttt{sculptured}.\texttt{Artwork} \sqcap \forall \texttt{created}.\texttt{Sculpture}$

3.2.2 Semantics

- Mathematical meaning/interpretation of syntax
- The interpretation function $\mathcal{I} = \left(\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}}\right)$ where $\Delta^{\mathcal{I}}$ is a non-empty domain of individuals, and $\cdot^{\mathcal{I}}$ is an interpretation function that maps
  - $A^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$, i.e. concepts to subsets of $\Delta^{\mathcal{I}}$
  - $r^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$, i.e. role names to subsets of $\Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$
- $\Delta^{\mathcal{I}}$ has to be not empty as otherwise no model exists.
- Thus, a concept/class/type represents a set of individuals: $\mathcal{I}(\texttt{Painter}) = \{\texttt{rembrandt}, \texttt{vanGogh}\}$
- Interpretation of a role/relation: $\mathcal{I}(\texttt{hasPainted}) = \{(\texttt{rembrandt}, \texttt{nightwatch}), (\texttt{daVinci}, \texttt{MonaLisa})\}$
- $\cdot^{\mathcal{I}}$ is inductively extended over complex concept descriptions

$$
\begin{aligned}
\top^{\mathcal{I}} &= \Delta^{\mathcal{I}} \\
\bot^{\mathcal{I}} &= \emptyset \\
(\neg C)^{\mathcal{I}} &= \Delta^{\mathcal{I}} \setminus C^{\mathcal{I}} \\
(C \sqcap D)^{\mathcal{I}} &= C^{\mathcal{I}} \cap D^{\mathcal{I}} \\
(C \sqcup D)^{\mathcal{I}} &= C^{\mathcal{I}} \cup D^{\mathcal{I}} \\
(\exists r.C)^{\mathcal{I}} &= \left\{x \in \Delta^{\mathcal{I}} \mid \exists y.(x,y) \in r^{\mathcal{I}} \wedge y \in C^{\mathcal{I}}\right\} \\
(\forall r.C)^{\mathcal{I}} &= \left\{x \in \Delta^{\mathcal{I}} \mid \forall y.(x,y) \in r^{\mathcal{I}} \rightarrow y \in C^{\mathcal{I}}\right\}
\end{aligned}
$$

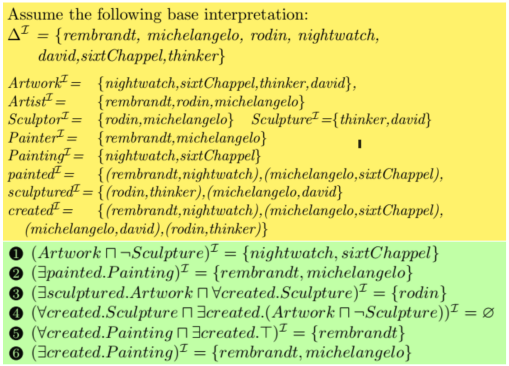


Figure 8: Example for interpretation function $\mathcal{I}$

- Note that the quantifier $\forall r.C$ also includes the empty set. To prevent this, use $\forall r.C \sqcap \exists r.\top$

3.3 Inference in Description Logic

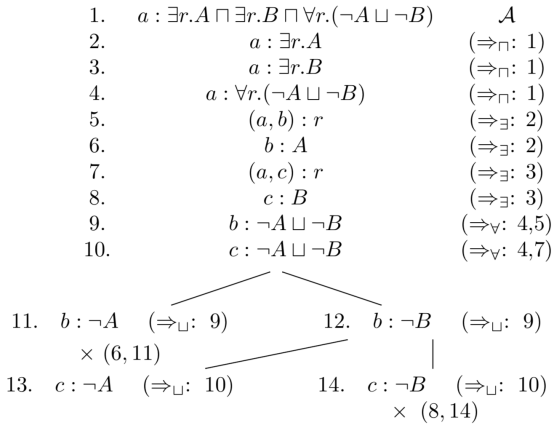
- A **knowledge base** $\mathcal{K} = (\mathcal{T}, \mathcal{A})$ consists of:
  - the terminology/theory $\mathcal{T}$, that defines the general world model you can apply to any model. It summarizes the relations between concepts.
    - The knowledge about relations between concepts is expressed by means of terminological axioms
    - *Concept inclusion*: $C \sqsubseteq D$ models necessary conditions for object of type C
      Example: $\texttt{Elephant} \sqsubseteq \texttt{Animal} \sqcap \neg\texttt{Mouse}$
    - *Concept equivalence*: $C \equiv D$ models necessary and sufficient conditions for object C. Can also be written as $C \sqsubseteq D$ and $D \sqsubseteq C$
      Example: $\mathcal{T}$: "*painter is a human that created a painting*" $\Rightarrow \texttt{Painter} \equiv \texttt{human} \sqcap \exists \texttt{created.Painting}$
  - the assertions $\mathcal{A}$ that describes certain objects and the sets that are defined by $\mathcal{T}$
    - Knowledge about individuals in the domain expressed in terms of the vocabulary is specified by means of assertional axioms
    - *Concept assertion*: $a : C$ models that individual a is of class C
      Example: $\texttt{dumbo} : \texttt{Elephant}$
    - *Role assertions*: $(a, b) : r$ models that individual a is related to individual b by the role r
      Example: $(\texttt{daVinci}, \texttt{MonaLisa}) : \texttt{painted}$
- Definition of a ***model***
  - An interpretation $\mathcal{I}$ is a *model* of the $\mathcal{T}$-box iff it satisfies every terminological axiom in $\mathcal{T}$
  - An interpretation $\mathcal{I}$ is a *model* of the $\mathcal{A}$-box iff it satisfies every assertional axiom in $\mathcal{A}$
    Note that this just requires $\mathcal{I}$ to have the individuals and relations/assertions defined in $\mathcal{A}$
  - Finally, an interpretation $\mathcal{I}$ is a *model* of the knowledge base $\mathcal{K} = (\mathcal{T}, \mathcal{A})$-box iff $\mathcal{I}$ is a model of both $\mathcal{T}$ and $\mathcal{A}$
  - A knowledge base is called *satisfiable/consistent* iff there exists a model for it.
- In general, axioms of $\mathcal{A}$- and $\mathcal{T}$-box restrict the possible models
- **Reasoning tasks** for $\mathcal{T}$-box
  - *Concept satisfiability*: Concept C is satisfiable w.r.t. $\mathcal{T}$ iff there is a model $\mathcal{I}$ of $\mathcal{T}$: $C^{\mathcal{I}} \neq \emptyset$
  - *Subsumption*: check if $\mathcal{T} \models C \sqsubseteq D$. C is subsumed by D in $\mathcal{T}$ iff $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$ in every model $\mathcal{I}$ of $\mathcal{T}$
  - *Equivalence*: check if $\mathcal{T} \models C \equiv D$. Concepts C and D are equivalent in $\mathcal{T}$ iff $C^{\mathcal{I}} = D^{\mathcal{I}}$ in every model $\mathcal{I}$ of $\mathcal{T}$
  - All $\mathcal{T}$-box problems can be reduced to concept satisfiability which is equivalent to showing that $C \sqcap \neg D$ is unsatisfiable in $\mathcal{T}$ (see section 3.3.1 for Tableau algorithm to solve this)
- **Reasoning tasks** for $\mathcal{A}$-box
  - *$\mathcal{A}$-box consistency*: $\mathcal{A}$ is consistent w.r.t. $\mathcal{T}$ iff there is a model of $\mathcal{K}$. In such a case, $\mathcal{K}$ is satisfiable.
  - *Instance checking*: check if $\mathcal{K} \models a : C$ (or respectively $\mathcal{K} \models (a, b) : r$). This holds iff for every model of $\mathcal{K}$ is a model of $a : C$
  - Question: don't we also need the assertion that $\mathcal{K}$ is consistent as otherwise we can prove anything for an unsatisfiable knowledge base?
  - *Retrieval task*: given a concept C and an $\mathcal{A}$-box $\mathcal{A}$, find all individuals a such that $\mathcal{K} \models a : C$
  - *Realization task*: given an individual a and a set of concepts, find the most specific concept C such that $\mathcal{K} \models a : C$
  - All tasks are reducible to checking $\mathcal{A}$-box consistency. We can check those by showing that $\mathcal{A} \cup \{a : \neg C\}$ is inconsistent.

3.3.1 Tableau algorithm

- All reasoning tasks in $\mathcal{ALC}$ can be reduced to a single task of checking $\mathcal{A}$-box consistency w.r.t. $\mathcal{T}$-box.
- Tableau shows (un-)satisfiability by contradiction. It searches through the tree of possible models and delivers a proof iff no model exists and therefore the input inconsistent is (or otherwise returns a valid model).
- The general approach of Tableau is to extend the model until we find a model, or closed all branches
- To reduce the number of tableau proves, we assume that all concepts appear in the Negation Normal Form (NNF). Conversion rules:

$$\begin{aligned} \neg\,(C \sqcap D) &\Rightarrow (\neg C \sqcup \neg D) \\ \neg\,(C \sqcup D) &\Rightarrow (\neg C \sqcap \neg D) \\ \neg\exists r.C &\Rightarrow \forall r.\neg C \\ \neg\forall r.C &\Rightarrow \exists r.\neg C \end{aligned}$$

- A ***branch*** of a tableau is a set of $\mathcal{A}$-box assertions. For any branch S, the following rules apply:
  - **IF** $(a : C \sqcap D) \in S$ **THEN** $S' := S \cup \{a : C, a : D\}$
  - **IF** $(a : C \sqcup D) \in S$ **THEN** $S' := S \cup \{a : C\}$ **or** $S' := S \cup \{a : D\}$
  - **IF** $(a : \exists r.C) \in S$ **THEN** $S' := S \cup \{(a, b) : r, b : C\}$ where b is a "fresh" individual name in S
  - **IF** $(a : \forall r.C) \in S$ **and** $(a, b) : r \in S$ **THEN** $S' := S \cup \{b : C\}$
  - **IF** $\{a : C, a : \neg C\} \subseteq S$ **THEN** mark branch as CLOSED (unsatisfiable)
- To check for satisfiability, we assume that there is an individual of that given concept. Try to show that this leads to an contradiction.
- Example: *show that* $\exists r.A \sqcap \exists r.B$ *is subsumed by* $\exists r.(A \sqcap B)$
  1. Write expression as concept: $\exists r.A \sqcap \exists r.B \sqsubseteq \exists r.(A \sqcap B)$
  2. Negate concept (proof by contradiction): $\exists r.A \sqcap \exists r.B \sqcap \neg\exists r.(A \sqcap B)$
  3. Rewrite concept in NNF: $\exists r.A \sqcap \exists r.B \sqcap \forall r.(\neg A \sqcup \neg B)$
  4. Assume $\mathcal{A}$-box $\mathcal{A} = \{a : \exists r.A \sqcap \exists r.B \sqcap \forall r.(\neg A \sqcup \neg B)\}$ and search for a model



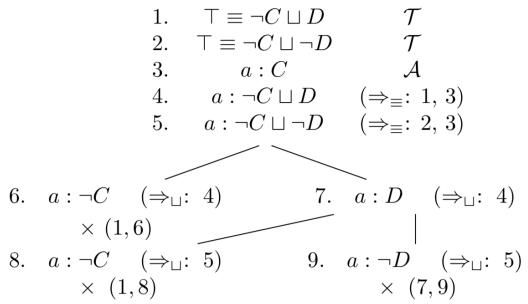
  5. Step 13 shows the valid model where $\Delta^{\mathcal{I}} = \{a, b, c\}$, $A^{\mathcal{I}} = \{b\}$, $B^{\mathcal{I}} = \{c\}$, $r^{\mathcal{I}} = \{(a, b), (a, c)\} \Rightarrow$ the negated concept is satisfiable and thus proves the hypothesis to be wrong
     If we would not have been able to construct a model, the hypothesis would be correct
- Note that the tableau algorithm is sound (if algorithm finds a proof, then statement is correct) and complete (if statement is correct, the algorithm finds a proof).

3.3.2 Reasoning with non-empty $\mathcal{T}$-box

- The tableau algorithm can be extended in order to support terminology axioms of the $\mathcal{T}$-box
- Input preprocessing
  - Replace every $C \equiv D \in \mathcal{T}$ with $C \sqsubseteq D$ and $D \sqsubseteq C$
  - Replace every $C \sqsubseteq D \in \mathcal{T}$ with $\top \equiv NNF(\neg C \sqcup D)$
  - Add all concepts/formula of $\mathcal{T}$ to the root S_0 of the tableau
- Extend the tableau rules by the following:
  - **IF** $(\top \equiv C) \in S$ **and** an individual a occurs in S **THEN** $S' := S \cup \{a : C\}$
- Example:

  Problem: Is C satisfiable w.r.t. $\mathcal{T} = \{C \sqsubseteq D, C \sqsubseteq \neg D\}$?

  Tableau proof:



  Output: C is **UNSATISFIABLE** w.r.t. $\mathcal{T}$

- In case of reasoning with non-empty $\mathcal{T}$-box, the tableau algorithm is not guaranteed to terminate (rules in the form of $\top \equiv ...\exists r.C...$ can lead to an infinite loop)
- Solution: Detect cycles and prevent further application of the $\Rightarrow_\exists$ rule. This is achieved by a special blocking rule:
  - **IF** b is a (possibly indirect) successor of a in S **and** it is the case that:

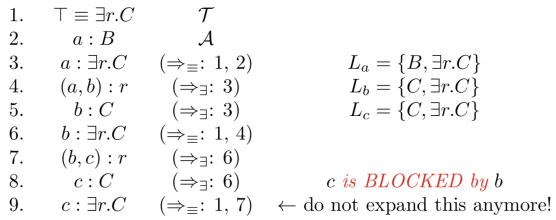
$$\{C \mid b : C \in S\} \subset \{D \mid a : D \in S\}$$

  THEN mark b as BLOCKED by a in S and do not apply $\Rightarrow_\exists$ rule to b.

- Example:

  Problem: Is B satisfiable w.r.t. $\mathcal{T} = \{\top \equiv \exists r.C\}$?

  Tableau proof:



  Output: C is **SATISFIABLE** w.r.t. $\mathcal{T}$

- Note that we can only apply the blocking rule if no other rules apply anymore on that branch.

4 Constraint Satisfaction Problems

- Knowledge Representation is focused on qualitative reasoning, not really quantitative
  - Abstract description of the world is mostly easier for human to reason than an exact numerical definition
  - We can for example reason about time points and intervals in our system where we have relations between those

4.1 Fundamentals of CSPs

- To represent qualitative knowledge, we again have to define a:
  - *Vocabulary*: finite set of relations, mostly binary.
    Example: x equals $y \Rightarrow x = y$, x before $y \Rightarrow x < y$, x after $y \Rightarrow x > y$
  - *Language*: sets of atomic formulae, perhaps restricted disjunction
    Example: define disjunction by $x \{<, =\} y$ (in maths $x \leq y$), and formulae to describe configurations: $\{x \{<, =\} y, y \{z\}\}$
  - *Formal semantics*: interpretation of function and symbols
    Example: interpret time points and relations over rational (or real) numbers
- On those, we can perform various reasoning tasks
  - *Satisfiability*: Is this a consistent set of constraints? $\Rightarrow$ find satisfying instantiation of all variables
  - *Deduction*: Does $x \{=\} y$ logically follow from the configuration?
  - *Minimal description*: What are the most constrained relations that describe the same set?
  - *Solving*: find one or all (optimal) solutions to the CSP
- Formally, a Constraint Satisfaction Problem consists of:
  - Variables $Y := y_1, y_2, ..., y_k$
  - Domains $D_1, ..., D_k$ to which the variables belong (y_i represents a possible value of D_i: $y_i \in D_i$)
  - Constraints $C \in \mathcal{C}$ on Y which define a subset of $D_1 \times D_2 \times ... \times D_k$
- Given a CSP $\{\mathcal{C}; y_1 \in D_1, ..., y_k \in D_k\}$, $(d_1, ..., d_k) \in D_1 \times ... \times D_k$ is a solution iff for all $C \in \mathcal{C}$: $(d_1, ..., d_k) \in C$
- Most CSPs have only binary constraints (constraints over two variables) which can be modeled by a constraint graph (nodes are variables, edges/arcs are constraints)

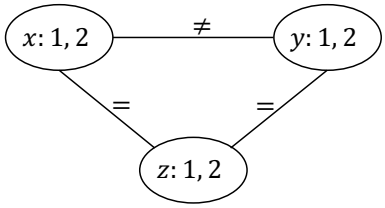
4.1.1 Backtrack search

- Finding a solution for CSPs is a more general version of DPLL (PL is actually a special case CSP)
- In the initial state, we have an empty set of assignments. We then assign a value to an unassigned variable that does not conflict with the current assignment. Test if CSP is fulfilled or unsatisfied for this assignment. If neither of both, choose the next variable etc.
- The depth of the search tree is the number of variables, and the number of children/splits in the tree are given by the domain size. In the worst case, we have a search space of $|D|^{\#vars}$
- We can improve backtracking efficiency by:
  1. **Smart variable picking**: various heuristics for choosing the next variable
     - *Most constrained variable*: choose the variable with the fewest legal values (therefore also called *Minimum remaining values* (MRV))
     - *Most constraining variable*: the variable with the most constraints on. Is mostly used as tie-breaker strategy for MRV
  2. **Smart value picking**: what possible value of the domain to pick for the variable
     - *Least constraining value*: given a variable, choose the value that constraints the least, i.e. the one that rules out the fewest values in the remaining values

3. **Spotting failure early** to backtrack early: finding unsatisfiable constraints in a branch
   - *Forward checking*: by keeping track of remaining legal values for unassigned variables, we can stop the search in this branch if one variable has an empty set/no legal values to be assigned. This is done by propagating information from assigned to unassigned variables.
   - *Arc consistency*: simplest form of constraint propagation (see Section 4.2) that makes each arc/edge consistent. Hence, $x \to y$ is consistent iff for every value x in D_x there exists a value y in D_y for which the constraint is satisfied (if not, remove contradicting values of D_x). Note that if x has been changed, we have to recheck all its neighbors. Arc consistency can detect failures earlier than forward checking
   - More complex/sophisticated methods are discussed under constraint propagation in Section 4.2

4.2 Constraint propagation

- A constraint itself can restrict the search space without splitting
- Constraint propagation is similar to unit propagation and subsumes resolution. It therefore checks the CSP for local consistency (a CSP is locally consistent if we can extend it by another assignment while being satisfied)
- Example: the CSP $\langle x < y; x \in [50..200], y \in [0..100] \rangle$ can be simplified to $\langle x < y; x \in [50..99], y \in [51..100] \rangle$ without losing any possible solutions
- This might be an iterative problem when multiple constraints interact with each other (x has effect on y, y on z, and so on)
- Thus, we have to decide when to stop performing local consistency checking. There are various heuristics/methods:
  - *(Directional) Arc consistency* is one of the most simplest approaches. Note that general arc consistency applies propagation to both sides $x \to y$ and $y \to x$, while we can also limit it by a directed approach (only $x \to y$ or $y \to x$). Problem: only looks at direct constraints and can for example not spot that the following CSP is inconsistent:



  - *Path consistency*: extends arc consistency to picking two variables. Formally, for all i and j:

$$\forall x, y : (x, y) \in P_{i,j}, x \in P_i, y \in P_j \to (\exists z : z \in P_k \wedge (x, z) \in P_{i,k} \wedge (z, y) \in P_{k,j})$$

    i.e., any consistent assignment to two variables can be extended to a third one. Can be enforced by iterating the following assignment until nothing changes anymore:

$$P_{i,j} := P_{i,j} \cap \{(x, y) \mid \exists z : (x, z) \in P_{i,k} \wedge (z, y) \in P_{k,j} \text{ for all } k\}$$

    Path consistency subsumes arc consistency and detects inconsistency in the previous example. Note that path consistency removes pairs of values, and thus makes constraints explicitly
  - k-*Consistency*: generalization of arc and path consistency to arbitrary k. For each satisfying value assignment to $k-1$ variables, there exists an extension of this assignment to a k-th variable such that this extended assignment satisfies all constraints among these k variables.
  - *Strong k-consistency*: A CSP is strongly k-consistent iff it is j-consistent for all $j \leq k$. If a CSP of size n n-consistent is, then we can construct a model in polynomial time. But note that checking this already solves the CSP and the computational costs exponentially grow with k.
  - Question: why does PC subsumes AC, but not k-consistency $k-1$-consistency?
  - *Hyper-arc consistency*: extension of arc consistency to more than binary constraints (same for those). Defined by: for every constraint C and every variable x with domain D_x, each value for x from D_x participates in a solution to C.

5 Qualitative Reasoning

- Learning by making (qualitative) representations, combine meanings and reason with them
- Differs with numerical reasoning by not having exact values (we can express that something is positive, but we don't say it has the value 3.21)

5.1 General vocabulary

- In QR, we express systems by the use of entities (like *population*), and describe them by their quantities (e.g. *size*)
- We can define relations and inequalities between those. The full vocabulary is visualized in Figure 9

Entity	
Quantity	
Relationship structural	Relation
Change (dec/std/inc)	▼ Ø ▲
Proportional (neg/pos)	P- P+
Influence (neg/pos)	I- I+
Inequality	≤, <, =, >, ≥
Subtraction (addition)	A-B=C, A+B=C

Figure 9: Overview of the vocabulary used in QR

5.1.1 Quantities

- Quantities are described by a magnitude and a derivative (can also include higher order derivatives)
- The quantities are represented by a discrete scales which are build up by
  - *Intervals* which defines a range of values (e.g. $+$ for positive values)
  - *Landmarks* which reflect key points of the system, and are defined as single number (e.g. 0 or $\max$)

5.1.2 Influence and Proportional

- **Proportional**: If $\mathtt{Q1}$ increases, then $\mathtt{Q2}$ increases (P+)/decreases (P−): $\partial \mathtt{Q1} > 0 \implies \partial \mathtt{Q2} > 0$
  - Mathematically, we can define the relation by:

$$\mathtt{Q2} \propto_{Q+} \mathtt{Q1} \quad \equiv \quad \exists f \mid \mathtt{Q2} = f(..., \mathtt{Q1}, ...) \wedge f \text{ is increasing monotonic in } \mathtt{Q1}$$

  - This means that if $\mathtt{Q2}$ is *qualitatively* proportional to $\mathtt{Q1}$, we can express $\mathtt{Q2}$ by a function based on $\mathtt{Q1}$ such that it increases if $\mathtt{Q1}$ increases (and everything else stays the same)
  - Monotonic functions provide an abstraction that cover a wide range of more mathematical expressions which can be narrowed by further constraints
- **Influence**: If $\mathtt{Q1}$ is greater than 0, then $\mathtt{Q2}$ increases (I+)/decreases (I−): $\mathtt{Q1} > 0 \implies \partial \mathtt{Q2} > 0$
  - Expressed in mathematical terms, the influence relation is stated as:

$$I^+(\mathtt{Q2}, \mathtt{Q1}) \equiv d\mathtt{Q2}/dt = ... + B + ...$$

  - Note that this function definition is much more specific than for proportional. Hence it enables us knowledge of relative rates if multiple influences are given. Example: if $I^+(\mathtt{Q2}, \mathtt{Q1}) \wedge I^-(\mathtt{Q2}, \mathtt{Q3})$, then we know that if $\mathtt{Q1} > \mathtt{Q3}$, $\mathtt{Q2}$ increases.
  - *Warning*: for reasoning with influences, we often require the second-order derivative (if $\mathtt{Q1}$ has a derivative and doesn't change its qualitative magnitude, we can't model the change in the derivative of $\mathtt{Q2}$). This is especially important if we have a positive and negative influence which both are in a interval with their magnitude.
- Example in formulas:

- The change of the size of a closed population can be specified by: $N_{t+1} = N_t + B_t - D_t$ where B is the number of births, and D the number of deaths between t and $t + 1$.
- Then, we have a positive influence of B to N, and a negative influence of D to N: $I^+(N, B)$, $I^-(N, D)$
- Further, we specify that the birth and death rate proportionally depend on the population size: $B = f_B(..., N, ...)$, $D = f_D(..., N...) \Rightarrow P^+(D, N)$, $P^+(B, N)$

- Both relations can lead to ambiguity if one influence/proportion is positive, and another is negative. Without given any other facts, all outcomes are possible new states

5.1.3 Inequalities

- We can define inequalities between quantities (magnitudes), landmarks and derivatives
- Inequalities state a rather uncertain knowledge, or assumptions that are valid for at least the initial state
- For example, we can state that A is larger than B, but this might change during the simulation
- Also, we can reason with inequalities by propagating knowledge (if $A > B$ and $B > C$, then $A > C$). This can for example lead to constraints in the value space of another quantity

5.1.4 Value constraint

- The value constraint binds landmarks/intervals of two quantities together.
- If we have a value constraint from A to B to both 0, this means that if $A = 0$ then $B = 0$ (directed relation!)

5.2 States and Transitions

- A qualitative state is a period of time in which the qualitative behavior of the system doesn't change (changes include magnitudes, derivatives, inequalities, etc.)
- Thus, states are *unique* sets of inequality quantity expressions
- Transitions are changes of at least one quantity (but all changes must take place to enter the next state if multiple quantities change)

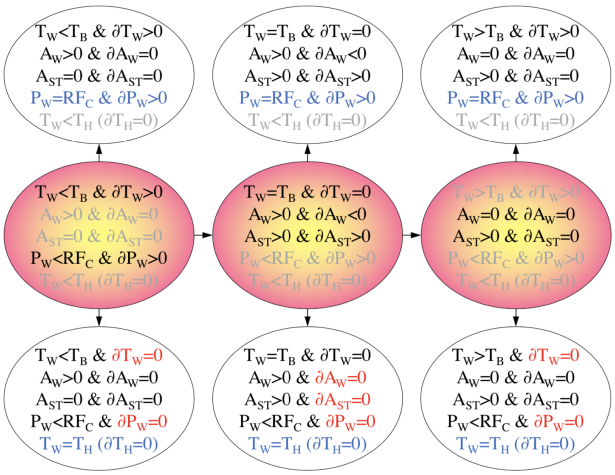


Figure 10: Example state graph for heating water in a cattle

5.2.1 Compositional modeling

- To improve re-usability of QR definitions, we can split the model into several parts
- We define a hierarchy of generic entities (e.g. population) on which we will operate

- The library of model fragments contain structures that defines certain entities/quantities and their relation (if a population exists, then...). These can be conditioned on assumptions (e.g. closed or open population)
- In the scenarios, we define certain instances of our entity hierarchy (e.g. "*Green Frogs*" as instance of entity population). Based on them, we can state initial values (e.g. population size is max), and assumptions we make (e.g. closed population)
- We then combine both the scenario and the model fragment to create the behavior graph

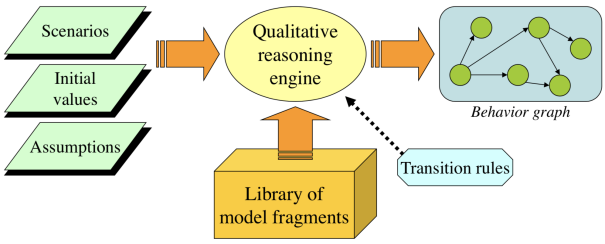


Figure 11: Compositional modeling

5.2.2 Finding transitions

- There are three common types of termination
  - **Value termination**: if a quantity has a derivative in a certain direction, then it can move to the next quantity point in this direction
  - **Inequality termination**: Given a inequality, if one side is changing due to derivatives, then the truth value might change (from $\texttt{Q2} > \texttt{Q1}$ to $\texttt{Q2} = \texttt{Q1}$, remember continuous changes!)
  - **Exogenous termination**: An external effect/input that leads to a change. Can for example be controlling the derivative of a quantity. The behavior can be random, sinusoidal, random, etc.
- Another important concept for deciding on transitions is **Epsilon ordering** dealing with value termination
  - We distinguish between *immediate* and *non-immediate* transitions
  - Immediate transitions are if a derivative is unequals 0 and the magnitude is currently at a landmark/point. Then, we will immediately move to the next state with the magnitude being changed
  - Non-immediate transitions are when the derivative is unequals zero, but the magnitude is in an interval. Then it can change at some point, but does not have to be
- Another concept we have to consider is value correspondence
  - If two values are restraint by a value correspondence, then this can limit the possible transitions we can have
  - For example, in Figure 12, we can either apply `T1` or `T2`, but both can't happen at the same time because then the value correspondence is not fulfilled

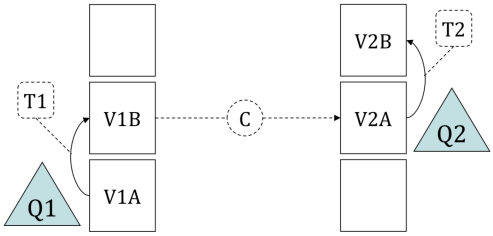


Figure 12: Example for transitions that exclude each other because of a value constraint