Decision Procedures for Verification

Combinations of Decision Procedures (4)

13.02.2017

Viorica Sofronie-Stokkermans

sofronie@uni-koblenz.de

Last time

From conjunctions to arbitrary combinations

Known:

Methods for checking satisfiability for conjunctions of literals

Question:

how to check satisfiability of sets of clauses?

Overview

Satisfiability w.r.t. theories

- Propositional logic
  - resolution
  - DPLL
- First-order logic
  - resolution

- Ground formulae
  - conjunctions of literals: specialized methods
  - clauses: DPLL(T) $\Leftarrow$ ctd. Today
- Formulae with quantifiers
  - reduction to SAT for ground formulae instantiation $\Leftarrow$ Idea: Today (situations when sound and complete)
  - resolution (mod T)

3.6 The $DPLL(\mathcal{T})$ algorithm

SAT Modulo Theories (SMT)

"Lazy" approaches to SMT: Idea

Example: consider $\mathcal{T} = UIF$ and the following set of clauses:

$$\underbrace{f(g(a)) \not\approx f(c)}_{\neg P_1} \vee \underbrace{g(a) \approx d}_{P_2}, \quad \underbrace{g(a) \approx c}_{P_3}, \quad \underbrace{c \not\approx d}_{\neg P_4}$$

1. Send $\{\neg P_1 \vee P_2,\ P_3,\ \neg P_4\}$ to SAT solver

   SAT solver returns model $[\neg P_1, P_3, \neg P_4]$
   Theory solver says $\neg P_1 \wedge P_3 \wedge \neg P_4$ is $\mathcal{T}$-inconsistent

2. Send $\{\neg P_1 \vee P_2,\ P_3,\ \neg P_4,\ P_1 \vee \neg P_3 \vee P_4\}$ to SAT solver

   SAT solver returns model $[P_1, P_2, P_3, \neg P_4]$
   Theory solver says $P_1 \wedge P_2 \wedge P_3 \wedge \neg P_4$ is $\mathcal{T}$-inconsistent

3. Send $\{\neg P_1 \vee P_2, P_3, \neg P_4, P_1 \vee \neg P_3 \vee P_4, \neg P_1 \vee \neg P_2 \vee \neg P_3 \vee P_4\}$ to SAT solver

   SAT solver says UNSAT

SAT Modulo Theories (SMT)

Optimized lazy approach

LA	• Check T-consistency only of full propositional models
OLA	• Check T-consistency of partial assignment while being built
LA	• Given a T-inconsistent assignment M, add $\neg M$ as a clause
OLA	• Given a T-inconsistent assignment M, find an explanation (a small T-inconsistent subset of M) and add it as a clause
LA	• Upon a T-inconsistency, add clause and restart
OLA	• Upon a T-inconsistency, do conflict analysis of the explanation and Backjump

SAT Modulo Theories (SMT)

“Lazy” approaches to SMT

- Why “lazy”?

Theory information used only lazily, when checking $\mathcal{T}$-consistency of propositional models

- Characteristics:

  + Modular and flexible
  – Theory information does not guide the search (only validates a posteriori)

Tools: CVC-Lite, ICS, MathSAT, TSAT+, Verifun, ...

“Lazy” approaches to SMT

Lazy theory learning:

$$M, L, M_1 || F \Rightarrow \emptyset || F, \neg L_1 \vee \cdots \vee \neg L_n \vee \neg L \quad \text{if} \begin{cases} M, L, M_1 \models F \\ \{L_1, \ldots, L_n\} \subseteq M \\ L_1 \wedge \cdots \wedge L_n \wedge L \models_{\mathcal{T}} \bot \end{cases}$$

Lazy theory learning + no repetitions

$$M, L, M_1 || F \Rightarrow \emptyset || F, \neg L_1 \vee \cdots \vee \neg L_n \vee \neg L \quad \text{if} \begin{cases} \{L_1, \ldots, L_n\} \subseteq M \\ L_1 \wedge \cdots \wedge L_n \wedge L \models_{\mathcal{T}} \bot \\ \neg L_1 \vee \cdots \vee \neg L_n \vee \neg L \notin F \end{cases}$$

DPLL(T) Rules

UnitPropagation

$M || F, C \vee L \Rightarrow M, L || F, C \vee L$ if $M \models \neg C$, and L undef. in M

Decide

$M || F \Rightarrow M, L^d || F$ if L occurs in F, L undef. in M

Fail

$M || F, C \Rightarrow \text{Fail}$ if $M \models \neg C$, no backtrack possible

Backjump

$M, L^d, N || F \Rightarrow M, L' || F$ if $\begin{cases} \text{there is some clause } C \vee L' \text{ s.t.:} \\ F \models C \vee L', M \models \neg C, \\ L' \text{ undefined in } M \\ L' \text{ or } \neg L' \text{ occurs in } F. \end{cases}$

Restart/Learn

$M || F \Rightarrow \emptyset || F, F'$ if $F \models F'$, F' obtained from M, F

TPropagation

$M || F \Rightarrow M, L || F$ if $M \models_{\mathcal{T}} L$

DPLL(T) Example

Consider again same example with UIF:

$$\underbrace{f(g(a)) \not\approx f(c)}_{\neg P_1} \vee \underbrace{g(a) \approx d}_{P_2}, \quad \underbrace{g(a) \approx c}_{P_3}, \quad \underbrace{c \not\approx d}_{\neg P_4}$$

$\emptyset$	$\|\|\neg P_1 \vee P_2, P_3, \neg P_4$	$\Rightarrow (UnitPropagation)$
P_3	$\|\|\neg P_1 \vee P_2, P_3, \neg P_4$	$\Rightarrow (TPropagation)$
$P_3 P_1$	$\|\|\neg P_1 \vee P_2, P_3, \neg P_4$	$\Rightarrow (UnitPropagation)$
$P_3 P_1 P_2$	$\|\|\neg P_1 \vee P_2, P_3, \neg P_4$	$\Rightarrow (TPropagation)$
$P_3 P_1 P_2 P_4$	$\|\|\neg P_1 \vee P_2, P_3, \neg P_4$	$\Rightarrow fail$

No search in this example

Termination

Idea: $DPLL(T)$ terminates if no clause is learned infinitely many times, since only finitely many such new clauses (built over input literals) exist.

Theorem. There exists no infinite sequence of the form

$$\emptyset || F \Rightarrow S_1 \Rightarrow S_2 \ldots$$

if no clause C is learned by Reset & Learn/Lazy Theory Learning infinitely many times along a sequence.

A similar termination result holds also for the DPLL(T) approach with Theory Propagation.

Termination

Theorem. There exist no infinite sequences of the form $\emptyset || F \Rightarrow S_1 \Rightarrow S_2 \ldots$

Proof. (Idea) We define a well-founded strict partial ordering $\succ$ on states, and show that each rule application $M||F \Rightarrow M'||F'$ is decreasing with respect to this ordering, i.e., $M||F \succ M'||F'$.

Let M be of the form $M_0, L_1, M_1, \ldots L_p, M_p$, where $L_1, \ldots, L_p$ are all the decision literals of M. Similarly, let M' be $M'_0, L'_1, M'_1, \ldots L'_{p'}, M'_{p'}$.

Let N be the number of distinct atoms (propositional variables) in F.

(Note that p, p' and the length of M and M' are always smaller than or equal to N.)

Termination

Theorem. There exist no infinite sequences of the form $\emptyset || F \Rightarrow S1 \Rightarrow \ldots$

Proof. (continued)
Let $m(M)$ be $N -$ length(M) (nr. of literals missing in M for M to be total).

Define: $M_0 L_1 M_1 \ldots L_p M_p || F \succ M_0' L_1' M_1' \ldots L_{p'}' M_{p'}' || F'$ if

(i) there is some i with $0 \leq i \leq p, p'$ such that

$m(M_0) = m(M_0'), \ldots m(M_{i-1}) = m(M_{i-1}'), m(M_i) > m(M_i')$ or

(ii) $m(M_0) = m(M_0'), \ldots m(M_p) = m(M_p')$ and $m(M) > m(M')$.

Comparing the number of missing literals in sequences is a strict ordering (irreflexive and transitive) and it is well-founded, and hence this also holds for its lexicographic extension on tuples of sequences of bounded length.

No learning/forgetting: It is easy to see that all Basic DPLL rule applications are decreasing with respect to $\succ$ if fail is added as an additional minimal element. (The rules UnitPropagate and Backjump decrease by case (i) of the definition and Decide decreases by case (ii).)

Termination

Theorem. There exist no infinite sequences of the form $\emptyset || F \Rightarrow S1 \Rightarrow \ldots$

Note: Combine learning with basic DPLL(T): no clause learned infinitely many times.

Forget: For this termination condition to be fulfilled, applying at least one rule of the Basic DPLL system between any two Learn applications does not suffice. It suffices if, in addition, no clause generated with Learning is ever forgotten.

Soundness, Correctness, Termination

Lemma. If $\emptyset || F \Rightarrow^* M || F'$ then:

(1) All atoms in M and all atoms in F' are atoms of F.

(2) M: no literal more than once, no complementary literals

(3) F' is logically equivalent to F

(4) if $M = M_0 L_1 M_1 \dots L_n M_n$ where L_i all decision literals
then $F, L_1, \dots, L_i \models M_i$.

Lemma. If $\emptyset || F \Rightarrow^* M || F'$, where $M || F'$ is a final state wrt the Basic DPLL system and Lazy Theory Learning, then:

(1) All literals of F' are defined in M

(2) There is no clause C in F' such that $M \models \neg C$

(3) M is a model of F.

Soundness, Correctness, Termination

Lemma. If $\emptyset || F \Rightarrow^* M || F'$, where $M || F'$ is a final state wrt the Basic DPLL system and Lazy Theory Learning, then M is a $\mathcal{T}$-model of F.

Theorem. The Lazy Theory learning DPLL system provides a decision procedure for the satisfiability in $\mathcal{T}$ of CNF formulae F, that is:

1. $\emptyset || F \Rightarrow^*$ *fail* if, and only if, F is unsatisfiable in $\mathcal{T}$.
2. $\emptyset || F \Rightarrow^* M || F'$, where $M || F'$ is a final state wrt the Basic DPLL system and Lazy Theory Learning, if, and only if, F is satisfiable in $\mathcal{T}$.

Proof

(1) If $\emptyset || F \Rightarrow^*$ *fail* then there exists state $M || F'$ with $\emptyset || F \Rightarrow^* M || F' \Rightarrow$ *fail*, there is no decision literal in M and $M \models \neg C$ for some clause C in F. By the construction of M, $F \models M$, so $F \models \neg C$. Thus F is unsatisfiable.

To prove the converse, if $\emptyset || F \not\Rightarrow^*$ *fail* then by there must be a state $M || F'$ such that $\emptyset || F \Rightarrow^* M || F'$. Then $M \models F$, so F is satisfiable.

Soundness, Correctness, Termination

Lemma. If $\emptyset || F \Rightarrow^* M || F'$, where $M || F'$ is a final state wrt the Basic DPLL system and Lazy Theory Learning, then M is a $\mathcal{T}$-model of F.

Theorem. The Lazy Theory learning DPLL system provides a decision procedure for the satisfiability in $\mathcal{T}$ of CNF formulae F, that is:

1. $\emptyset || F \Rightarrow^* \textit{fail}$ if, and only if, F is unsatisfiable in $\mathcal{T}$.
2. $\emptyset || F \Rightarrow^* M || F'$, where $M || F'$ is a final state wrt the Basic DPLL system and Lazy Theory Learning, if, and only if, F is satisfiable in $\mathcal{T}$.

Proof

2. If $\emptyset || F \Rightarrow^* M || F$ then F is satisfiable. Conversely, if $\emptyset || F \not\Rightarrow^* M || F$ then $\emptyset || F \Rightarrow^* \textit{fail}$, so F is unsatisfiable.

Termination, Soundness and Completeness

DPLL($\mathcal{T}$) with (eager) theory propagation

Lemma. If $\emptyset || F \Rightarrow M || F$ then M is $\mathcal{T}$-consistent.

Proof. This property is true initially, and all rules preserve it, by the fact that $M \models_{\mathcal{T}} L$ if, and only if, $M \cup \neg L$ is $\mathcal{T}$-inconsistent: the rules only add literals to M that are undefined in M, and Theory Propagate adds all literals L of F that are theory consequences of M, before any literal $\neg L$ making it $\mathcal{T}$-inconsistent can be added to M by any of the other rules.

Termination, Soundness and Completeness

DPLL($\mathcal{T}$) with (eager) theory propagation

Definition. A DPLL($\mathcal{T}$) procedure with Eager Theory Propagation for $\mathcal{T}$ is any procedure taking an input CNF F and computing a sequence $\emptyset || F \Rightarrow^* S$ where S is a final state wrt. Theory Propagate and the Basic DPLL system.

Theorem The DPLL system with eager theory propagation provides a decision procedure for the satisfiability in $\mathcal{T}$ of CNF formulae F, that is:

1. $\emptyset || F \Rightarrow^* \mathit{fail}$ if, and only if, F is unsatisfiable in $\mathcal{T}$.
2. $\emptyset || F \Rightarrow^* M || F'$, where $M || F'$ is a final state wrt the Basic DPLL system and Theory Propagate, if, and only if, F is satisfiable in $\mathcal{T}$.
3. If $\emptyset || F \Rightarrow M || F'$, where $M || F'$ is a final state wrt the Basic DPLL system and Theory Propagate, then M is a $\mathcal{T}$-model of F.

Literature

Full proofs and further details can be found in:

Robert Nieuwenhuis, Albert Oliveras and Cesare Tinelli:
"Solving SAT and SAT Modulo Theories: From an Abstract Davis-Putnam-Logemann-Loveland Procedure to DPLL(T)"

Journal of the ACM, Vol. 53, No. 6, November 2006, pp.937-977.

SMT tools

SAT problems

Given: conjunction ϕ of prop. clauses
Task: check if ϕ satisfiable

Method: DPLL

- deterministic choices first
  - unit resolution
  - pure literal assignment
- case distinction (splitting)
- heuristics
  - selection criteria for splitting
  - backtracking
  - conflict-driven learning

SMT tools

SAT problems

Given: conjunction ϕ of prop. clauses
Task: check if ϕ satisfiable

Method: DPLL

- deterministic choices first
  - unit resolution
  - pure literal assignment
- case distinction (splitting)
- heuristics
  - selection criteria for splitting
  - backtracking
  - conflict-driven learning

SMT problems

Given: conjunction ϕ of clauses
Task: check if $\phi \models_{\mathcal{T}} \perp$

Method: DPLL($\mathcal{T}$)

- Boolean assignment found
  using DPLL
- ... and checked for $\mathcal{T}$-satisfiability
- the assignment can be partial
  and checked before splitting
- usual heuristics are used:
  - non-chronological backtracking
  - learning

SMT tools

SAT problems

Given: conjunction ϕ of prop. clauses
Task: check if ϕ satisfiable

Method: DPLL

- deterministic choices first
  unit resolution
  pure literal assignment
- case distinction (splitting)
- heuristics
  selection criteria for splitting
  backtracking
  conflict-driven learning

SMT problems

Given: conjunction ϕ of clauses
Task: check if $\phi \models_{\mathcal{T}} \perp$

Method: DPLL($\mathcal{T}$)

- Boolean assignment found
  using DPLL
- ... and checked for $\mathcal{T}$-satisfiability
- the assignment can be partial
  and checked before splitting
- usual heuristics are used:
  non-chronological backtracking
  learning

Systems implementing such specialized satisfiability problems: Yices, Barcelogic Tools, CVC lite, haRVey, Math-SAT,... are called (S)atisfiability (M)odulo (T)heory solvers.

Satisfiability of formulae with quantifiers

Satisfiability of formulae with quantifiers

In many applications we are interested in testing the satisfiability of formulae containing (universally quantified) variables.

Examples

- check satisfiability of formulae in the Bernays-Schönfinkel class
- check whether a set of (universally quantified) Horn clauses entails a ground clause
- check whether a property is consequence of a set of axioms

  Example 1: $f : \mathbb{Z} \rightarrow \mathbb{Z}$ is monotonely increasing
  and $g : \mathbb{Z} \rightarrow \mathbb{Z}$ is defined by $g(x) = f(x + x)$
  then g is also monotonely increasing

  Example 2: If array a is increasingly sorted, and
  x is inserted before the first position i with $a[i] > x$
  then the array remains increasingly sorted.

A theory of arrays

We consider the theory of arrays in a many-sorted setting.

Syntax:

- Sorts: Elem (elements), Array (arrays) and Index (indices, here integers).
- Function symbols: read, write.

$a(\text{read}) = \mathit{Array} \times \mathit{Index} \rightarrow \mathit{Element}$

$a(\text{write}) = \mathit{Array} \times \mathit{Index} \times \mathit{Element} \rightarrow \mathit{Array}$

Theories of arrays

We consider the theory of arrays in a many-sorted setting.

Theory of arrays $\mathcal{T}_{arrays}$:

- $\mathcal{T}_i$ (theory of indices): Presburger arithmetic
- $\mathcal{T}_e$ (theory of elements): arbitrary
- Axioms for read, write

$$\begin{array}{rcl} read(write(a, i, e), i) & \approx & e \\ j \not\approx i \vee read(write(a, i, e), j) & = & read(a, j). \end{array}$$

Theories of arrays

We consider the theory of arrays in a many-sorted setting.

Theory of arrays $\mathcal{T}_{arrays}$:

- $\mathcal{T}_i$ (theory of indices): Presburger arithmetic
- $\mathcal{T}_e$ (theory of elements): arbitrary
- Axioms for read, write

$$\begin{aligned} read(write(a, i, e), i) &\approx e \\ j \not\approx i \vee read(write(a, i, e), j) &= read(a, j). \end{aligned}$$

Fact: Undecidable in general.

Goal: Identify a fragment of the theory of arrays which is decidable.

A decidable fragment

- **Index guard** a positive Boolean combination of atoms of the form $t \leq u$ or $t = u$ where t and u are either a variable or a ground term of sort Index

  Example: $(x \leq 3 \vee x \approx y) \wedge y \leq z$ is an index guard

  Example: $x + 1 \leq c, \quad x + 3 \leq y, \quad x + x \leq 2$ are not index guards.

- **Array property formula** [Bradley,Manna,Sipma'06]

  $(\forall i)(\varphi_I(i) \rightarrow \varphi_V(i))$, where:

  φ_I: index guard

  φ_V: formula in which any universally quantified i occurs in a direct array read; no nestings

  Example: $c \leq x \leq y \leq d \rightarrow a(x) \leq a(y)$ is an array property formula

  Example: $x < y \rightarrow a(x) < a(y)$ is not an array property formula

Decision Procedure

(Rules should be read from top to bottom)

Step 1: Put F in NNF.

Step 2: Apply the following rule exhaustively to remove writes:

$$\frac{F[write(a, i, v)]}{F[a'] \wedge a'[i] = v \wedge (\forall j. j \neq i \rightarrow a[j] = a'[j])} \quad \text{for fresh } a' \text{ (write)}$$

Given a formula F containing an occurrence of a write term $write(a, i, v)$, we can substitute every occurrence of $write(a, i, v)$ with a fresh variable a' and explain the relationship between a' and a.

Decision Procedure

Step 3 Apply the following rule exhaustively to remove existential quantification:

$$\frac{F[\exists i.\,G[i]]}{F[G[j]]} \text{ for fresh } j \text{ (exists)}$$

Existential quantification can arise during Step 1 if the given formula contains a negated array property.

Decision Procedure

Steps 4-6 accomplish the reduction of universal quantification to finite conjunction.

The main idea is to select a set of symbolic index terms on which to instantiate all universal quantifiers.

Theories of arrays

Step 4 From the output F3 of Step 3, construct the index set $\mathcal{I}$:

$$
\begin{aligned}
\mathcal{I} \quad = \quad & \{\lambda\} \cup \\
& \{t \mid \cdot[t] \in F3 \text{ such that } t \text{ is not a universally quantified variable}\} \cup \\
& \{t \mid t \text{ occurs as an } \textit{evar} \text{ in the parsing of index guards}\}
\end{aligned}
$$

(evar is any constant, ground term, or unquantified variable.)

This index set is the finite set of indices that need to be examined. It includes all terms t that occur in some $\textit{read}(a, t)$ anywhere in F (unless it is a universally quantified variable) and all terms t that are compared to a universally quantified variable in some index guard.

λ is a fresh constant that represents all other index positions that are not explicitly in $\mathcal{I}$.

Theories of arrays

Step 5 Apply the following rule exhaustively to remove universal quantification:

$$\frac{H[\forall \bar{i}. F[i] \rightarrow G[i]]}{H\left[\bigwedge_{\bar{i} \in \mathcal{I}^n} (F[\bar{i}] \rightarrow G[\bar{i}])\right]} \quad \text{(forall)}$$

where n is the size of the list of quantified variables $\bar{i}$.

This is the key step.

It replaces universal quantification with finite conjunction over the index set. The notation $\bar{i} \in \mathcal{I}^n$ means that the variables $\bar{i}$ range over all n-tuples of terms in $\mathcal{I}$.

Theories of arrays

Step 6: From the output $F5$ of Step 5, construct

$$F6 : \quad F5 \wedge \bigwedge_{i \in \mathcal{I} \setminus \{\lambda\}} \lambda \neq i$$

The new conjuncts assert that the variable λ introduced in Step 4 is unique: it does not equal any other index mentioned in F5.

Step 7: Decide the TA-satisfiability of $F6$ using the decision procedure for the quantifier free fragment.

Example

Consider the array property formula

$$F : write(a, l, v)[k] = b[k] \wedge b[k] \neq v \wedge a[k] = v \wedge (\forall i. i \neq l \rightarrow a[i] = b[i])$$

It contains one array property,

$$\forall i. i \neq l \rightarrow a[i] = b[i]$$

index guard: $i \neq l := (i \leq l - 1 \vee i \geq l + 1)$ value constraint: $a[i] = b[i]$

Step 1: The formula is already in NNF.

Step 2: We rewrite F as:

$$\begin{aligned} F2 : \quad & a'[k] = b[k] \wedge b[k] \neq v \wedge a[k] = v \wedge (\forall i. i \neq l \rightarrow a[i] = b[i]) \\ & \wedge a'[l] = v \wedge (\forall j. j \neq l \rightarrow a[j] = a'[j]). \end{aligned}$$

Example

Consider the array property formula

$$F : write(a, l, v)[k] = b[k] \wedge b[k] \neq v \wedge a[k] = v \wedge (\forall i . i \neq l \rightarrow a[i] = b[i])$$

Step 2: We rewrite F as:

$$F2 : \quad a'[k] = b[k] \wedge b[k] \neq v \wedge a[k] = v \wedge (\forall i . i \neq l \rightarrow a[i] = b[i]) \\ \wedge a'[l] = v \wedge (\forall j . j \neq l \rightarrow a[j] = a'[j]).$$

index guards: $i \neq l := (i \leq l - 1 \vee i \geq l + 1)$ value constraint: $a[i] = b[i]$

$i \neq l := (j \leq l - 1 \vee j \geq l + 1)$ value constraint: $a[i] = a'[j]$

Step 3: F2 does not contain any existential quantifiers $\mapsto$ F3 = F2.

Step 4: The index set is

$$\mathcal{I} = \{\lambda\} \cup \{k\} \cup \{l, l-1, l+1\} = \{\lambda, k, l, l-1, l+1\}$$

Example

Consider the array property formula

$$F : write(a, l, v)[k] = b[k] \wedge b[k] \neq v \wedge a[k] = v \wedge (\forall i . i \neq l \rightarrow a[i] = b[i])$$

Step 3:

$$\begin{aligned} F3 : \quad & a'[k] = b[k] \wedge b[k] \neq v \wedge a[k] = v \wedge (\forall i . i \neq l \rightarrow a[i] = b[i]) \\ & \wedge a'[l] = v \wedge (\forall j . j \neq l \rightarrow a[j] = a'[j]). \end{aligned}$$

Step 4: $\mathcal{I} = \{\lambda, k, l, l - 1, l + 1\}$

Step 5: we replace universal quantification as follows:

$$\begin{aligned} F5 : \quad & a'[k] = b[k] \wedge b[k] \neq v \wedge a[k] = v \wedge \bigwedge_{i \in \mathcal{I}} (i \neq l \rightarrow a[i] = b[i]) \\ & \wedge a'[l] = v \wedge \bigwedge_{i \in \mathcal{I}} (j \neq l \rightarrow a[j] = a'[j]). \end{aligned}$$

Example

Consider the array property formula

$$F : write(a, l, v)[k] = b[k] \land b[k] \neq v \land a[k] = v \land (\forall i. i \neq l \rightarrow a[i] = b[i])$$

$$\mathcal{I} = \{\lambda, k, l, l-1, l+1\}$$

Step 5 (continued) Expanding produces:

$$\begin{aligned}
F5' : \quad & a'[k] = b[k] \land b[k] \neq v \land a[k] = v \land \\
& (\lambda \neq l \rightarrow a[\lambda] = b[\lambda]) \land (k \neq l \rightarrow a[k] = b[k]) \land (l \neq l \rightarrow a[l] = b[l]) \land \\
& (l-1 \neq l \rightarrow a[l-1] = b[l-1]) \land (l+1 \neq l \rightarrow a[l+1] = b[l+1]) \land \\
& a'[l] = v \land (\lambda \neq l \rightarrow a[\lambda] = a'[\lambda]) \land (k \neq l \rightarrow a[k] = a'[k]) \land \\
& (l \neq l \rightarrow a[l] = a'[l]) \land (l-1 \neq l \rightarrow a[l-1] = a'[l-1]) \land \\
& (l+1 \neq l \rightarrow a[l+1] = a'[l+1]).
\end{aligned}$$

Example

Consider the array property formula

$$F : write(a, l, v)[k] = b[k] \wedge b[k] \neq v \wedge a[k] = v \wedge (\forall i. i \neq l \rightarrow a[i] = b[i])$$

$$\mathcal{I} = \{\lambda\} \cup \{k\} \cup \{l\} = \{\lambda, k, l\}$$

Step 5 (continued): Simplifying produces

$$\begin{aligned} F''5 : \quad & a'[k] = b[k] \wedge b[k] \neq v \wedge a[k] = v \wedge (\lambda \neq l \rightarrow a[\lambda] = b[\lambda]) \\ & \wedge (k \neq l \rightarrow a[k] = b[k]) \wedge a[l-1] = b[l-1] \wedge a[l+1] = b[l+1] \\ & \wedge a'[l] = v \wedge (\lambda \neq l \rightarrow a[\lambda] = a'[\lambda]) \\ & \wedge (k \neq l \rightarrow a[k] = a'[k]) \wedge a[l-1] = a'[l-1] \wedge a[l+1] = a'[l+1]. \end{aligned}$$

Example

Consider the array property formula

$$F : write(a, l, v)[k] = b[k] \wedge b[k] \neq v \wedge a[k] = v \wedge (\forall i. i \neq l \rightarrow a[i] = b[i])$$

Step 6 distinguishes λ from other members of I:

$$\begin{aligned} F6 : \quad & a'[k] = b[k] \wedge b[k] \neq v \wedge a[k] = v \wedge (\lambda \neq l \rightarrow a[\lambda] = b[\lambda]) \\ & \wedge (k \neq l \rightarrow a[k] = b[k]) \wedge a[l-1] = b[l-1] \wedge a[l+1] = b[l+1] \\ & \wedge a'[l] = v \wedge (\lambda \neq l \rightarrow a[\lambda] = a'[\lambda]) \\ & \wedge (k \neq l \rightarrow a[k] = a'[k]) \wedge a[l-1] = a'[l-1] \wedge a[l+1] = a'[l+1] \\ & \wedge \lambda \neq k \wedge \lambda \neq l \wedge \lambda \neq l-1 \wedge \lambda \neq l+1. \end{aligned}$$

Example

Consider the array property formula

$$F : write(a, l, v)[k] = b[k] \wedge b[k] \neq v \wedge a[k] = v \wedge (\forall i. i \neq l \rightarrow a[i] = b[i])$$

Step 6 Simplifying, we have

$$\begin{aligned} F'6 : \quad & a'[k] = b[k] \wedge b[k] \neq v \wedge a[k] = v \wedge a[\lambda] = b[\lambda] \\ & \wedge a[k] = b[k] \wedge a[l-1] = b[l-1] \wedge a[l+1] = b[l+1] \\ & \wedge a'[l] = v \wedge a[\lambda] = a'[\lambda] \\ & \wedge (k \neq l \rightarrow a[k] = a'[k]) \wedge a[l-1] = a'[l-1] \wedge a[l+1] = a'[l+1] \\ & \wedge \lambda \neq k \wedge \lambda \neq l \wedge \lambda \neq l-1 \wedge \lambda \neq l+1. \end{aligned}$$

We can use for instance DPLL(T).
Alternative: Case distinction. There are two cases to consider.

(1) If $k{=}l$, then $a'[l]{=}v$ and $a'[k]{=}b[k]$ imply $b[k]{=}v$, yet $b[k]{\neq}v$.
(2) If $k{\neq}l$, then $a[k]{=}v$ and $a[k]{=}b[k]$ imply $b[k]{=}v$, but again $b[k]{\neq}v$.

Hence, F'6 is TA-unsatisfiable, indicating that F is TA-unsatisfiable.

Soundness and Completeness

Theorem (Soundness and Completeness)

Consider a formula F from the array property fragment . The output F6 of Step 6 is T_{arrays}-equisatisfiable to F.

Proof

(Soundness) Step 1-6 preserve satisfiability

(Fi is a logical consequence of F$i-1$).

Soundness and Completeness

Theorem (Soundness and Completeness)

Consider a formula F from the array property fragment . The output F6 of Step 6 is T_{arrays}-equisatisfiable to F.

Proof (Completeness)

Step 6: From the output $F5$ of Step 5, construct

$$F6 : \quad F5 \wedge \bigwedge_{i \in \mathcal{I} \setminus \{\lambda\}} \lambda \neq i$$

Assume that $F6$ is satisfiabile. Clearly F5 has a model.

Soundness and Completeness

Theorem (Soundness and Completeness)

Consider a formula F from the array property fragment . The output F6 of Step 6 is T_{arrays}-equisatisfiable to F.

Proof (Completeness)

Step 5 Apply the following rule exhaustively to remove universal quantification:

$$\frac{H[\forall \bar{i}. F[i] \rightarrow G[i]]}{H\left[\bigwedge_{\bar{i} \in \mathcal{I}^n}(F[\bar{i}] \rightarrow G[\bar{i}])\right]} \quad \text{(forall)}$$

Assume that $F5$ is satisfiabile. Let $\mathcal{A} = (\mathbb{Z}, \text{Elem}, \{a_A\}_{a \in Arrays}, ...)$ be a model for F5. Construct a model $\mathcal{B}$ for $F4$ as follows.

For $x \in \mathbb{Z}$: $l(x)$ $(u(x))$ closest left (right) neighbor of x in $\mathcal{I}$.

$$a_{\mathcal{B}}(x) = \begin{cases} a_{\mathcal{A}}(l(x)) & \textit{if } x - l(x) \leq u(x) - x \text{ or } u(x) = \infty \\ a_{\mathcal{A}}(u(x)) & \textit{if } x - l(x) > u(x) - x \text{ or } l(x) = -\infty \end{cases}$$

Soundness and Completeness

Theorem (Soundness and Completeness)

Consider a formula F from the array property fragment . The output F6 of Step 6 is T_{arrays}-equisatisfiable to F.

Proof (Completeness)

Step 3 Apply the following rule exhaustively to remove existential quantification:

$$\frac{F[\exists i.G[i]]}{F[G[j]]} \text{ for fresh } j \text{ (exists)}$$

If F3 has model then F2 has model

Soundness and Completeness

Theorem (Soundness and Completeness)

Consider a formula F from the array property fragment . The output F6 of Step 6 is T_{arrays}-equisatisfiable to F.

Proof (Completeness)

Step 2: Apply the following rule exhaustively to remove writes:

$$\frac{F[write(a, i, v)]}{F[a'] \wedge a'[i] = v \wedge (\forall j. j \neq i \rightarrow a[j] = a'[j])} \quad \text{for fresh } a' \text{ (write)}$$

Given a formula F containing an occurrence of a write term $write(a, i, v)$, we can substitute every occurrence of $write(a, i, v)$ with a fresh variable a' and explan the relationship between a' and a.

If F2 has a model then F1 has a model.

Step 1: Put F in NNF: NNF F1 is equivalent to F.

Theories of arrays

Theorem (Complexity) Suppose $(T_{index} \cup T_{elem})$-satisfiability is in NP. For sub-fragments of the array property fragment in which formulae have bounded-size blocks of quantifiers, T_{arrays}-satisfiability is NP-complete.

Proof NP-hardness is clear.

That the problem is in NP follows easily from the procedure: instantiating a block of n universal quantifiers quantifying subformula G over index set I produces $|I| \cdot n$ new subformulae, each of length polynomial in the length of G. Hence, the output of Step 6 is of length only a polynomial factor greater than the input to the procedure for fixed n.

Program verification

Example: Does BubbleSort return a sorted array?

$$
\begin{array}{l}
\text{int [] BubbleSort(int[] } a) \ \{ \\
\quad \text{int } i, j, t; \\
\quad \text{for } (i := |a| - 1; i > 0; i := i - 1) \ \{ \\
\quad\quad \text{for } (j := 0; j < i; j := j + 1) \ \{ \\
\quad\quad\quad \text{if } (a[j] > a[j+1])\{t := a[j]; \\
\quad\quad\quad\quad a[j] := a[j+1]; \\
\quad\quad\quad\quad a[j+1] := t\}; \\
\}\} \text{ return } a\}
\end{array}
$$

Program Verification

$-1 \le i < |a| \wedge$
partitioned$(a, 0, i, i+1, |a|-1) \wedge$
sorted$(a, i, |a|-1)$

$-1 \le i < |a| \wedge 0 \le j \le i \wedge$
partitioned$(a, 0, i, i+1, |a|-1) \wedge$
sorted$(a, i, |a|-1)$
partitioned$(a, 0, j-1, j, j)$ C_2

Example: Does BubbleSort return a sorted array?

```
int [] BubbleSort(int[] a) {
   int i, j, t;
   for (i := |a| − 1; i > 0; i := i − 1) {
      for (j := 0; j < i; j := j + 1) {
         if (a[j] > a[j + 1]){t := a[j];
                              a[j] := a[j + 1];
                              a[j + 1] := t};
}} return a}
```

Generate verification conditions and prove that they are valid

Predicates:

- sorted(a, l, u): $\forall i, j (l \le i \le j \le u \rightarrow a[i] \le a[j])$
- partitioned(a, l_1, u_1, l_2, u_2): $\forall i, j (l_1 \le i \le u_1 \le l_2 \le j \le u_2 \rightarrow a[i] \le a[j])$

Program Verification

$-1 \leq i < |a| \wedge$
partitioned$(a, 0, i, i+1, |a|-1) \wedge$
sorted$(a, i, |a|-1)$

$-1 \leq i < |a| \wedge 0 \leq j \leq i \wedge$
partitioned$(a, 0, i, i+1, |a|-1) \wedge$
sorted$(a, i, |a|-1)$
partitioned$(a, 0, j-1, j, j)$ C_2

Example: Does BubbleSort return a sorted array?

```
int [] BubbleSort(int[] a) {
  int i, j, t;
  for (i := |a| − 1; i > 0; i := i − 1) {
    for (j := 0; j < i; j := j + 1) {
      if (a[j] > a[j + 1]){t := a[j];
                           a[j] := a[j + 1];
                           a[j + 1] := t};
}} return a}
```

Generate verification conditions and prove that they are valid

Predicates:

- sorted(a, l, u): $\forall i, j (l \leq i \leq j \leq u \rightarrow a[i] \leq a[j])$
- partitioned(a, l_1, u_1, l_2, u_2): $\forall i, j (l_1 \leq i \leq u_1 \leq l_2 \leq j \leq u_2 \rightarrow a[i] \leq a[j])$

To prove: $C_2(a) \wedge \text{Update}(a, a') \rightarrow C_2(a')$

Another Situation

Insertion of an element c in a sorted array a of length n

for $(i := 1; i \leq n; i := i + 1)$ $\{$
    if $a[i] \geq c\{n := n + 1$
        for $(j := n; j > i; j := j - 1)\{a[i] := a[i - 1]\}$
        $a[i] := c$; return a
$\}\}$ $a[n + 1] := c$; return a

Task:

If the array was sorted before insertion it is sorted also after insertion.

$\text{Sorted}(a, n) \wedge \text{Update}(a, n, a', n') \wedge \neg\text{Sorted}(a', n') \models_{\mathcal{T}} \perp$?

Another Situation

Task:

If the array was sorted before insertion it is sorted also after insertion.

$$\text{Sorted}(a, n) \wedge \text{Update}(a, n, a', n') \wedge \neg\text{Sorted}(a', n') \models_{\mathcal{T}} \perp ?$$

Sorted(a, n)	$\forall i, j(1 \leq i \leq j \leq n \rightarrow a[i] \leq a[j])$
Update(a, n, a', n')	$\forall i((1 \leq i \leq n \wedge a[i] < c) \rightarrow a'[i] = a[i])$
	$\forall i((c \leq a(1) \rightarrow a'[1] := c)$
	$\forall i((a[n] < c \rightarrow a'[n+1] := c)$
	$\forall i((1 \leq i-1 \leq i \leq n \wedge a[i-1] < c \wedge a[i] \geq c) \rightarrow (a'[i] = c)$
	$\forall i((1 \leq i-1 \leq i \leq n \wedge a[i-1] \geq c \wedge a[i] \geq c \rightarrow a'[i] := a[i-1])$
	$n' := n + 1$
$\neg$Sorted(a', n')	$\exists k, l(1 \leq k \leq l \leq n' \wedge a'k] > a'[l])$

Beyond the array property fragment

Extension: New arrays defined by case distinction – Def(f')

$$\forall \overline{x}(\phi_i(\overline{x}) \rightarrow f'(\overline{x}){=}s_i(\overline{x})) \qquad i \in I, \text{ where } \phi_i(\overline{x}) \wedge \phi_j(\overline{x}) \models_{\mathcal{T}_0} \perp \text{ for } i{\neq}j \quad (1)$$

$$\forall \overline{x}(\phi_i(\overline{x}) \rightarrow t_i(\overline{x}){\leq}f'(\overline{x}){\leq}s_i(\overline{x})) \qquad i \in I, \text{ where } \phi_i(\overline{x}) \wedge \phi_j(\overline{x}) \models_{\mathcal{T}_0} \perp \text{ for } i{\neq}j \quad (2)$$

where s_i, t_i are terms over the signature Σ such that $\mathcal{T}_0 \models \forall \overline{x}(\phi_i(\overline{x}){\rightarrow}t_i(\overline{x}){\leq}s_i(\overline{x}))$ for all $i \in I$.

$\mathcal{T}_0 \subseteq \mathcal{T}_0 \wedge \text{Def}(f')$ has the property that for every set G of ground clauses in which there are no nested applications of f':

$$\mathcal{T}_0 \wedge \text{Def}(f') \wedge G \models \perp \quad \text{iff} \quad \mathcal{T}_0 \wedge \text{Def}(f')[G] \wedge G$$

(sufficient to use instances of axioms in Def(f') which are relevant for G)

• Some of the syntactic restrictions of the array property fragment can be lifted

Example

Insertion in an array

(on the blackboard)

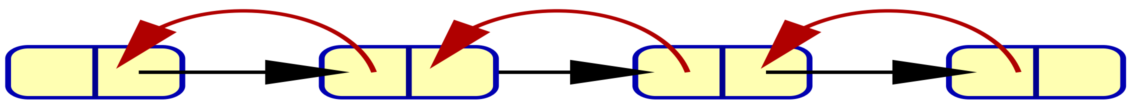
Pointer Structures

[McPeak, Necula 2005]

- pointer sort p, scalar sort s; pointer fields (p $\to$ p); scalar fields (p $\to$ s);
- axioms: $\forall p \ \ \mathcal{E} \vee \mathcal{C}$; $\mathcal{E}$ contains disjunctions of pointer equalities
  $\mathcal{C}$ contains scalar constraints

Assumption: If $f_1(f_2(\dots f_n(p)))$ occurs in axiom, the axiom also contains:
p=null $\vee$ $f_n(p)$=null $\vee \cdots \vee$ $f_2(\dots f_n(p)))$=null

Example: doubly-linked lists; ordered elements



$\forall p \ (p \neq \text{null} \wedge p.\text{next} \neq \text{null} \ \rightarrow \ p.\text{next}.\text{prev} = p)$

$\forall p \ (p \neq \text{null} \wedge p.\text{prev} \neq \text{null} \ \rightarrow \ p.\text{prev}.\text{next} = p)$

$\forall p \ (p \neq \text{null} \wedge p.\text{next} \neq \text{null} \ \rightarrow \ p.\text{info} \leq p.\text{next}.\text{info})$

Pointer Structures

[McPeak, Necula 2005]

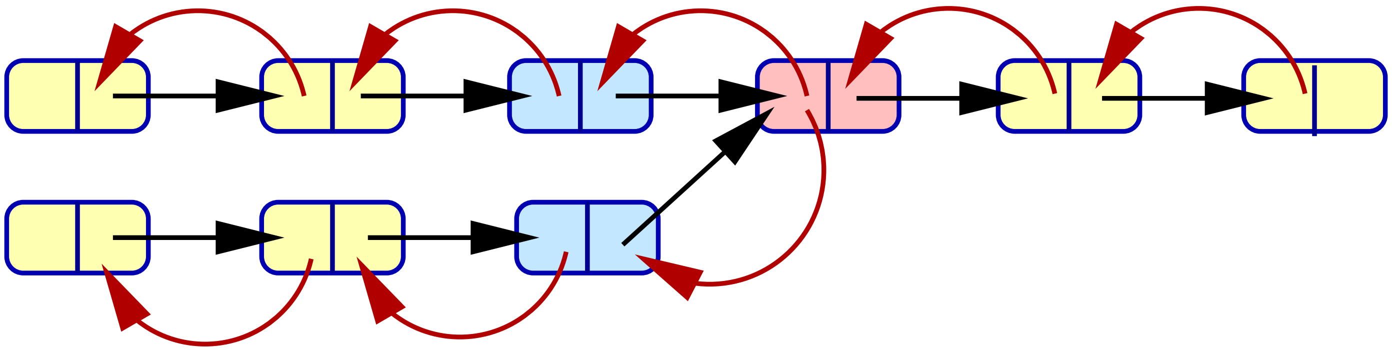
- pointer sort p, scalar sort s; pointer fields (p $\rightarrow$ p); scalar fields (p $\rightarrow$ s);
- axioms: $\forall p \;\; \mathcal{E} \vee \mathcal{C}$; $\quad$ $\mathcal{E}$ contains disjunctions of pointer equalities
  $\mathcal{C}$ contains scalar constraints

Assumption: If $f_1(f_2(\dots f_n(p)))$ occurs in axiom, the axiom also contains:
p=null $\vee$ $f_n(p)$=null $\vee \cdots \vee$ $f_2(\dots f_n(p)))$=null

Theorem. K set of clauses in the fragment above. Then for every set G of ground clauses, $(K \cup G) \cup \mathcal{T}_s \models \perp$ iff $K^{[G]} \cup \mathcal{T}_s \models \perp$

where $K^{[G]}$ is the set of instances of K in which the variables are replaced by subterms in G.

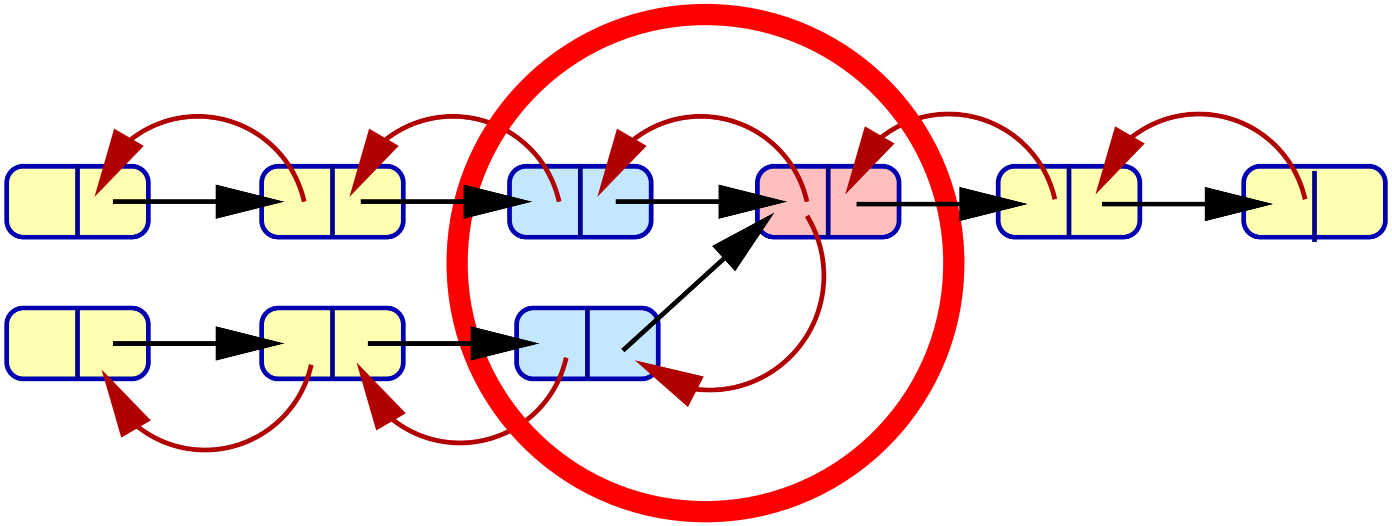
Example: A theory of doubly-linked lists



$$\forall p \; (p \neq \text{null} \wedge p.\text{next} \neq \text{null} \;\rightarrow\; p.\text{next}.\text{prev} = p)$$
$$\forall p \; (p \neq \text{null} \wedge p.\text{prev} \neq \text{null} \;\rightarrow\; p.\text{prev}.\text{next} = p)$$

$$\wedge \;\; c{\neq}\text{null} \wedge c.\text{next}{\neq}\text{null} \wedge d{\neq}\text{null} \wedge d.\text{next}{\neq}\text{null} \wedge c.\text{next}{=}d.\text{next} \wedge c \neq d \quad \models \quad \bot$$

Example: A theory of doubly-linked lists

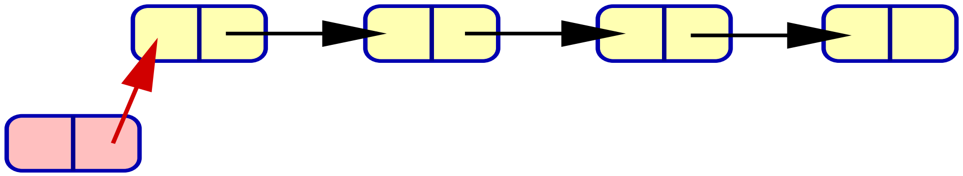


$(c \neq \text{null} \wedge c.\text{next} \neq \text{null} \rightarrow c.\text{next}.\text{prev} = c)$ $(c.\text{next} \neq \text{null} \wedge c.\text{next}.\text{next} \neq \text{null} \rightarrow c.\text{next}.\text{next}.\text{prev} = c.\text{nex}$

$(d \neq \text{null} \wedge d.\text{next} \neq \text{null} \rightarrow d.\text{next}.\text{prev} = d)$ $(d.\text{next} \neq \text{null} \wedge d.\text{next}.\text{next} \neq \text{null} \rightarrow d.\text{next}.\text{next}.\text{prev} = d.\text{nex}$

$$\wedge \quad c \neq \text{null} \wedge c.\text{next} \neq \text{null} \wedge d \neq \text{null} \wedge d.\text{next} \neq \text{null} \wedge c.\text{next} = d.\text{next} \wedge c \neq d \quad \models \quad \bot$$

Example: List insertion



Initially list is sorted: $p.\text{next} \neq \text{null} \rightarrow p.\text{prio} \geq p.\text{next.prio}$

$c.\text{prio} = x,\ c.\text{next} = \text{null}$

for all $p \neq c$ **do**

if $p.\text{prio} \leq x$ **then if** $\text{First}(p)$ **then** $c.\text{next}' = p,\ \text{First}'(c),\ \neg\text{First}'(p)$ **endif**; $p.\text{next}' = p.\text{next}$

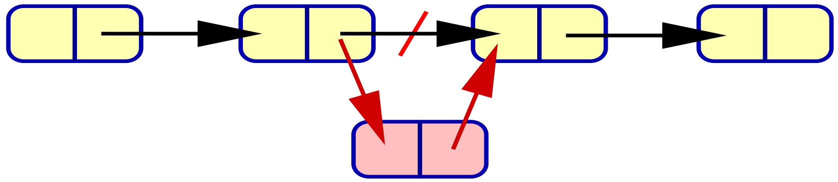
$p.\text{prio} > x$ **then** **case** $p.\text{next} = \text{null}$ **then** $p.\text{next}' := c,\ c.\text{next}' = \text{null}$

$p.\text{next} \neq \text{null} \wedge p.\text{next.prio} > x$ **then** $p.\text{next}' = p.\text{next}$

$p.\text{next} \neq \text{null} \wedge p.\text{next.prio} \leq x$ **then** $p.\text{next}' = c,\ c.\text{next}' = p.\text{nex}$

Verification task: After insertion list remains sorted

Example: List insertion



Initially list is sorted: $p.\text{next} \neq \text{null} \rightarrow p.\text{prio} \geq p.\text{next}.\text{prio}$

$c.\text{prio} = x, c.\text{next} = \text{null}$

for all $p \neq c$ **do**

if $p.\text{prio} \leq x$ **then if** $\text{First}(p)$ **then** $c.\text{next}' = p, \text{First}'(c), \neg\text{First}'(p)$ **endif**; $p.\text{next}' = p.\text{next}$

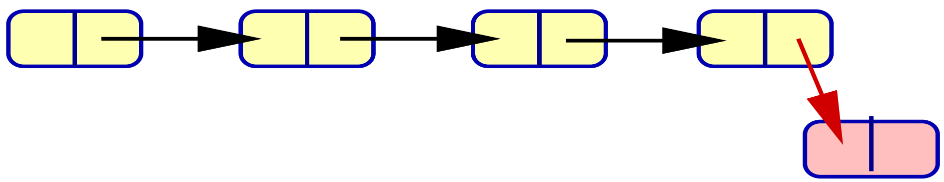
$p.\text{prio} > x$ **then** **case** $p.\text{next} = \text{null}$ **then** $p.\text{next}' := c, c.\text{next}' = \text{null}$

$p.\text{next} \neq \text{null} \wedge p.\text{next}.\text{prio} > x$ **then** $p.\text{next}' = p.\text{next}$

$p.\text{next} \neq \text{null} \wedge p.\text{next}.\text{prio} \leq x$ **then** $p.\text{next}' = c, c.\text{next}' = p.\text{next}$

Verification task: After insertion list remains sorted

Example: List insertion



Initially list is sorted: $p.\text{next} \neq \text{null} \rightarrow p.\text{prio} \geq p.\text{next.prio}$

$c.\text{prio} = x$, $c.\text{next} = \text{null}$

for all $p \neq c$ **do**

if $p.\text{prio} \leq x$ **then if** $\text{First}(p)$ **then** $c.\text{next}' = p$, $\text{First}'(c)$, $\neg\text{First}'(p)$ **endif**; $p.\text{next}' = p.\text{next}$

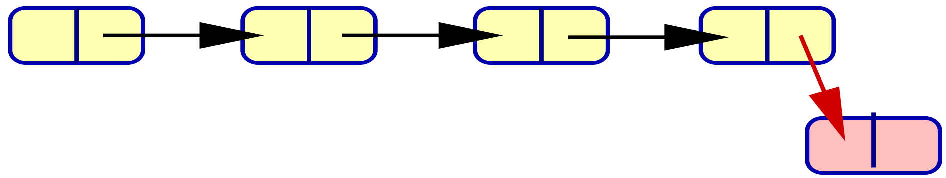
$p.\text{prio} > x$ **then** **case** $p.\text{next} = \text{null}$ **then** $p.\text{next}' := c$, $c.\text{next}' = \text{null}$

$p.\text{next} \neq \text{null} \wedge p.\text{next.prio} > x$ **then** $p.\text{next}' = p.\text{next}$

$p.\text{next} \neq \text{null} \wedge p.\text{next.prio} \leq x$ **then** $p.\text{next}' = c$, $c.\text{next}' = p.\text{next}$

Verification task: After insertion list remains sorted

Example: List insertion



Initially list is sorted: $\forall p(p.\text{next} \neq \text{null} \rightarrow p.\text{prio} \geq p.\text{next}.\text{prio})$

$\forall p(p{\neq}\text{null} \wedge p{\neq}c \wedge \text{prio}(p){\leq}x \wedge \text{First}(p) \rightarrow \text{next}'(c){=}p \wedge \text{First}'(c))$

$\forall p(p{\neq}\text{null} \wedge p{\neq}c \wedge \text{prio}(p){\leq}x \wedge \text{First}(p) \rightarrow \text{next}'(p){=}\text{next}(p) \wedge \neg\text{First}'(p))$

$\forall p(p{\neq}\text{null} \wedge p{\neq}c \wedge \text{prio}(p){\leq}x \wedge \neg\text{First}(p) \rightarrow \text{next}'(p){=}\text{next}(p))$

$\forall p(p{\neq}\text{null} \wedge p{\neq}c \wedge \text{prio}(p){>}x \wedge \text{next}(p){=}\text{null} \rightarrow \text{next}'(p){=}c$

$\forall p(p{\neq}\text{null} \wedge p{\neq}c \wedge \text{prio}(p){>}x \wedge \text{next}(p){=}\text{null} \rightarrow \text{next}'(c){=}\text{null})$

$\forall p(p{\neq}\text{null} \wedge p{\neq}c \wedge \text{prio}(p){>}x \wedge \text{next}(p){\neq}\text{null} \wedge \text{prio}(\text{next}(p)){>}x \rightarrow \text{next}'(p){=}\text{next}(p))$

$\forall p(p{\neq}\text{null} \wedge p$ … $(p){=}c$

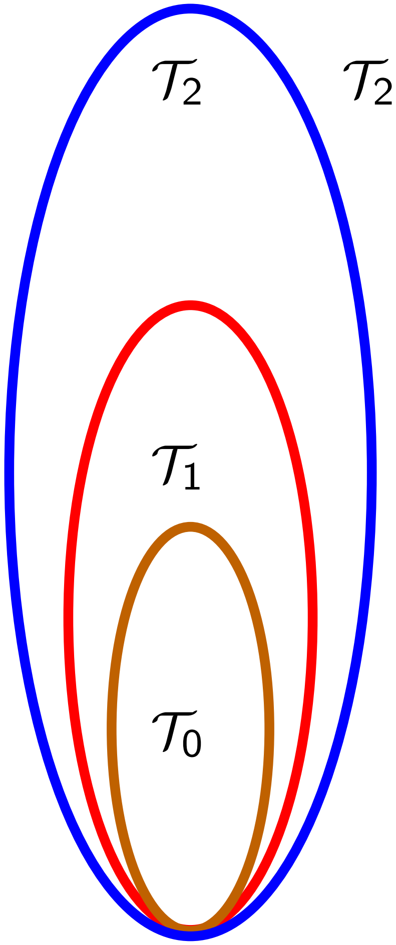
$\forall p(p{\neq}\text{null} \wedge p$ … $(c){=}\text{next}(p))$

We only need to use instances in which variables are replaced by ground subterms occurring in the problem

---

To check: $\text{Sorted}(\text{next}, \text{prio}) \wedge \text{Update}(\text{next}, \text{next}') \wedge p_0.\text{next}'{\neq}\text{null} {\wedge} p_0.\text{prio}{\not\geq} p_0.\text{next}'.\text{prio} \models \perp$

Example: List insertion



$\mathcal{T}_2 = \mathcal{T}_1 \cup \text{Update}(\text{next}, \text{next}')$

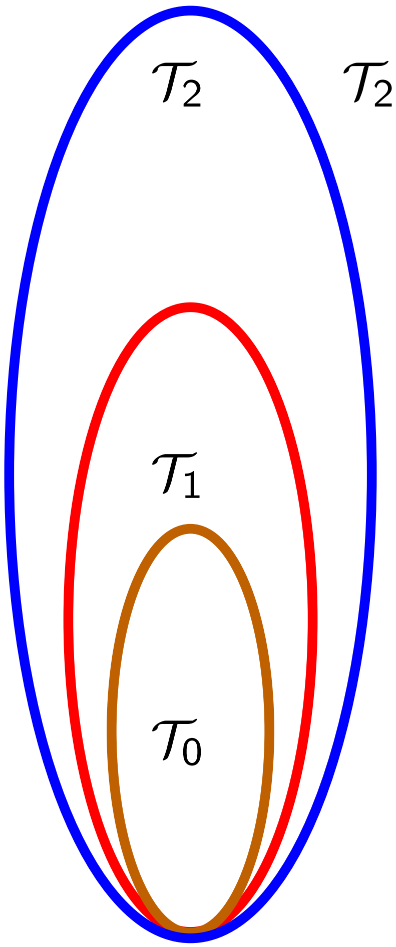
$\mathcal{T}_1 = \mathcal{T}_0 \cup \text{Sorted}(\text{next})$

$\mathcal{T}_0 = (\text{Lists}, \text{next})$

To show:

$\mathcal{T}_2 \cup \underbrace{\neg\text{Sorted}(\text{next}')}_{G} \models \perp$

Example: List insertion



$\mathcal{T}_2 = \mathcal{T}_1 \cup \boxed{\mathsf{Update}(\mathsf{next}, \mathsf{next}')}$

To show:

$\mathcal{T}_2 \cup \underbrace{\neg \mathsf{Sorted}(\mathsf{next}')}_{G} \models \perp$

Instantiate:

Hierarchical reasoning:

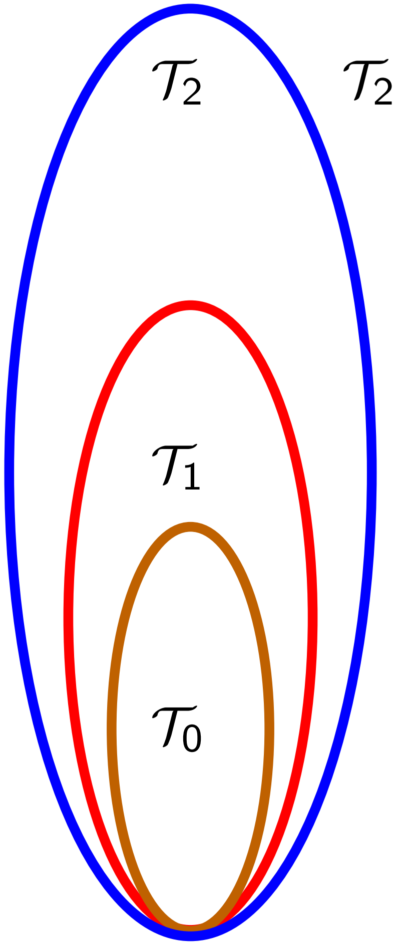
$\mathcal{T}_1 \cup \underbrace{\boxed{\mathsf{Update}(\mathsf{next}, \mathsf{next}')[G]} \cup G}_{G'}$

$\mathcal{T}_1 = \mathcal{T}_0 \cup \mathsf{Sorted}(\mathsf{next})$

$\mathcal{T}_1 \cup G'(\mathsf{next}) \models \perp$

$\mathcal{T}_0 = (\mathsf{Lists}, \mathsf{next})$

Example: List insertion



$$\mathcal{T}_2 = \mathcal{T}_1 \cup \text{Update}(\text{next}, \text{next}')$$

$$\mathcal{T}_1 = \mathcal{T}_0 \cup \text{Sorted}(\text{next})$$

$$\mathcal{T}_0 = (\text{Lists}, \text{next})$$

To show:

$$\mathcal{T}_2 \cup \underbrace{\neg\text{Sorted}(\text{next}')}_{G} \models \perp$$

$$\Downarrow$$

$$\mathcal{T}_1 \cup G'(\text{next}) \models \perp$$

$$\Downarrow$$

$$\mathcal{T}_0 \cup G'' \models \perp$$

More general concept

Local Theory Extensions

Satisfiability of formulae with quantifiers

Goal: generalize the ideas for extensions of theories

Example: Strict monotonicity

$$\mathbb{R} \cup \mathbb{Z} \cup \mathsf{Mon}(f) \cup \underbrace{(a < b \wedge f(a) = f(b) + 1)}_{G} \models \perp$$

$$\mathsf{Mon}(f) \qquad \forall i, j (i < j \rightarrow f(i) < f(j))$$

Problems:

- A prover for $\mathbb{R} \cup \mathbb{Z}$ does not know about f
- A prover for first-order logic may have problems with the reals and integers
- DPLL(T) cannot be used (Mon, $\mathbb{Z}, \mathbb{R}$: non-disjoint signatures)
- SMT provers may have problems with the universal quantifiers

Our goal: reduce search: consider certain instances $\mathsf{Mon}(f)[G]$
without loss of completeness

hierarchical/modular reasoning:
reduce to checking satisfiability of a set of constraints over $\mathbb{R} \cup \mathbb{Z}$

Local theory extensions

Solution: Local theory extensions

$\mathcal{K}$ set of equational clauses; $\mathcal{T}_0$ theory; $\mathcal{T}_1 = \mathcal{T}_0 \cup \mathcal{K}$

(Loc) $\mathcal{T}_0 \subseteq \mathcal{T}_1$ is local, if for ground clauses G,
$\mathcal{T}_0 \cup \mathcal{K} \cup G \models \perp$ iff $\mathcal{T}_0 \cup \mathcal{K}[G] \cup G$ has no (partial) model

Various notions of locality, depending of the instances to be considered: stable locality, order locality; extended locality.

Example: Strict monotonicity

$$\mathbb{R} \cup \mathbb{Z} \cup \text{Mon}(f) \cup \underbrace{(a < b \wedge f(a) = f(b) + 1)}_{G} \models \perp$$

Base theory ($\mathbb{R} \cup \mathbb{Z}$)	Extension
$a < b$	$f(a) = f(b) + 1$
	$\forall i, j(i < j \rightarrow f(i) < f(j))$

Example: Strict monotonicity

$$\mathbb{R} \cup \mathbb{Z} \cup \text{Mon}(f) \cup \underbrace{(a < b \wedge f(a) = f(b) + 1)}_{G} \models \perp$$

Extension is local $\mapsto$ replace axiom with ground instances

Base theory ($\mathbb{R} \cup \mathbb{Z}$)	Extension
$a < b$	$f(a) = f(b) + 1$
	$a < b \rightarrow f(a) < f(b)$
	$b < a \rightarrow f(b) < f(a)$

Solution 1:

$SMT(\mathbb{R} \cup \mathbb{Z} \cup UIF)$

Example: Strict monotonicity

$$\mathbb{R} \cup \mathbb{Z} \cup \text{Mon}(f) \cup \underbrace{(a < b \wedge f(a) = f(b) + 1)}_{G} \models \perp$$

Extension is local $\mapsto$ replace axiom with ground instances

Add congruence axioms. Replace pos-terms with new constants

Base theory ($\mathbb{R} \cup \mathbb{Z}$)	Extension
$a < b$	$f(a) = f(b) + 1$
	$a < b \rightarrow f(a) < f(b)$
	$b < a \rightarrow f(b) < f(a)$
	$a = b \rightarrow f(a) = f(b)$

Solution 2:

Hierarchical reasoning

Example: Strict monotonicity

$$\mathbb{R} \cup \mathbb{Z} \cup \mathsf{Mon}(f) \cup \underbrace{(a < b \wedge f(a) = f(b) + 1)}_{G} \models \perp$$

Extension is local $\mapsto$ replace axiom with ground instances

Replace f-terms with new constants

Add definitions for the new constants

Base theory ($\mathbb{R} \cup \mathbb{Z}$)	Extension
$a < b$	$a_1 = b_1 + 1$
	$a < b \rightarrow a_1 < b_1$
	$b < a \rightarrow b_1 < a_1$
	$a = b \rightarrow a_1 = b_1$

Example: Strict monotonicity

$$\mathbb{R} \cup \mathbb{Z} \cup \text{Mon}(f) \cup \underbrace{(a < b \wedge f(a) = f(b) + 1)}_{G} \models \perp$$

Extension is local $\mapsto$ replace axiom with ground instances

Replace f-terms with new constants

Add definitions for the new constants

Base theory ($\mathbb{R} \cup \mathbb{Z}$)	Extension
$a < b$	$a_1 = f(a)$
$a_1 = b_1 + 1$	$b_1 = f(b)$
$a < b \rightarrow a_1 < b_1$	
$b < a \rightarrow b_1 < a_1$	
$a = b \rightarrow a_1 = b_1$	

Reasoning in local theory extensions

Locality: $\mathcal{T}_0 \cup \mathcal{K} \cup G \models \perp$ iff $\mathcal{T}_0 \cup \mathcal{K}[G] \cup G \models \perp$

Problem: Decide whether $\mathcal{T}_0 \cup \mathcal{K}[G] \cup G \models \perp$

Solution 1: Use $SMT(\mathcal{T}_0{+}UIF)$: possible only if $\mathcal{K}[G]$ ground

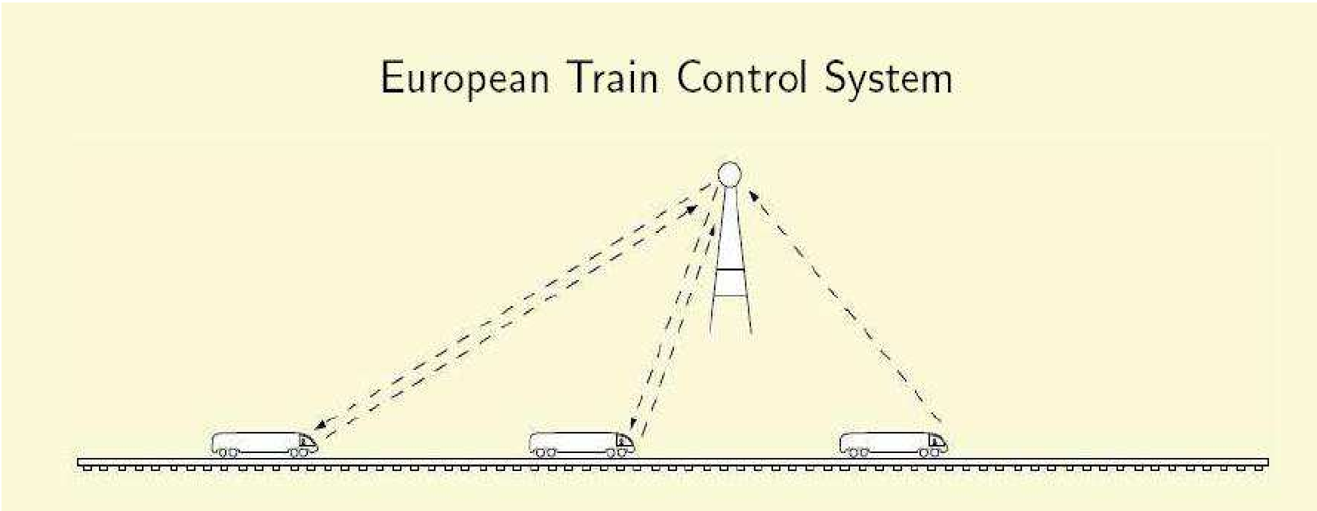
Solution 2: Hierarchic reasoning [VS'05]

reduce to satisfiability in $\mathcal{T}_0$: applicable in general

$\mapsto$ parameterized complexity

Example

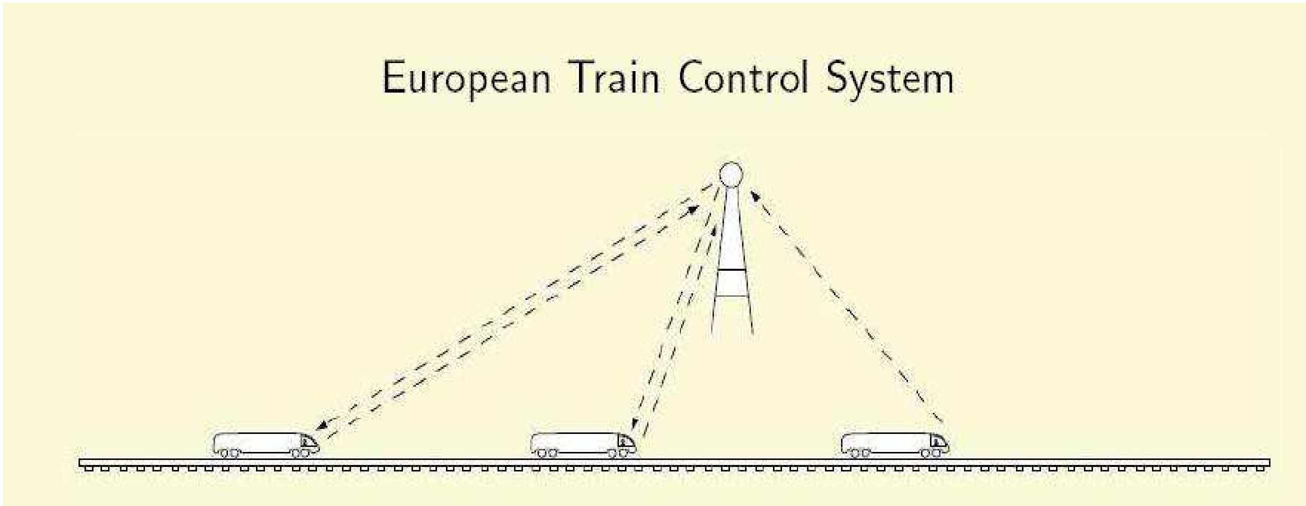
Simplified version of ETCS Case Study [Jacobs,VS'06, Faber,Jacobs,VS'07]



Number of trains:	$n \geq 0$	$\mathbb{Z}$
Minimum and maximum speed of trains:	$0 \leq \mathsf{min} < \mathsf{max}$	$\mathbb{R}$
Minimum secure distance:	$l_{\mathsf{alarm}} > 0$	$\mathbb{R}$
Time between updates:	$\Delta t > 0$	$\mathbb{R}$
Train positions before and after update:	$pos(i), pos'(i)$	$: \mathbb{Z} \rightarrow \mathbb{R}$

Example

Simplified version of ETCS Case Study [Jacobs,VS'06, Faber,Jacobs,VS'07]



Update(pos, pos′) :

- $\forall i \; (i = 0 \rightarrow pos(i) + \Delta t{*}\min \leq pos'(i) \leq pos(i) + \Delta t{*}\max)$
- $\forall i \; (0 < i < n \;\wedge\; pos(i-1) > 0 \;\wedge\; pos(i-1) - pos(i) \geq l_{\text{alarm}}$
  $\rightarrow pos(i) + \Delta t * \min \leq pos'(i) \leq pos(i) + \Delta t{*}\max)$

...

Example

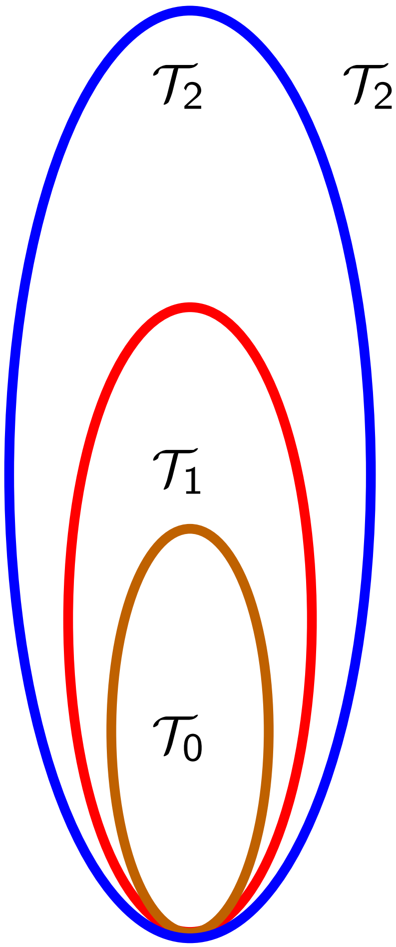
Safety property: No collisions $\mathsf{Safe}(\mathsf{pos}): \quad \forall i, j (i < j \rightarrow \mathsf{pos}(i) > \mathsf{pos}(j))$

Inductive invariant: $\mathsf{Safe}(\mathsf{pos}) \wedge \mathsf{Update}(\mathsf{pos}, \mathsf{pos}') \wedge \neg \mathsf{Safe}(\mathsf{pos}') \models_{\mathcal{T}_S} \perp$

where $\mathcal{T}_S$ is the extension of the (disjoint) combination $\mathbb{R} \cup \mathbb{Z}$ with two functions, $\mathsf{pos}, \mathsf{pos}' : \mathbb{Z} \rightarrow \mathbb{R}$

Our idea: Use chains of "instantiation" + reduction.

Example



$\mathcal{T}_2 = \mathcal{T}_1 \cup \mathsf{Update}(\mathsf{pos}, \mathsf{pos}')$

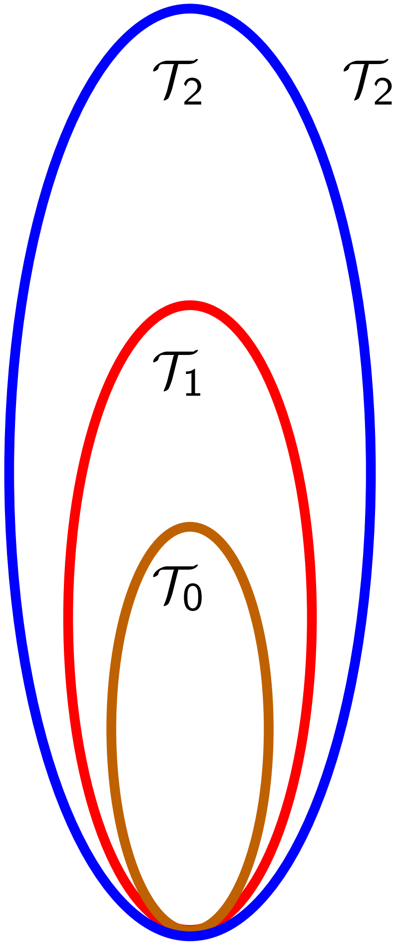
$\mathcal{T}_1 = \mathcal{T}_0 \cup \mathsf{Safe}(\mathsf{pos})$

$\mathcal{T}_0 = \mathbb{R} \cup \mathbb{Z}$

To show:

$$\mathcal{T}_2 \cup \underbrace{\neg \mathsf{Safe}(\mathsf{pos}')}_{G} \models \perp$$

Example



$\mathcal{T}_2 = \mathcal{T}_1 \cup \text{Update}(\text{pos}, \text{pos}')$

$\mathcal{T}_1 = \mathcal{T}_0 \cup \text{Safe}(\text{pos})$

$\mathcal{T}_0 = \mathbb{R} \cup \mathbb{Z}$

To show:

$$\mathcal{T}_2 \cup \underbrace{\neg\text{Safe}(\text{pos}')}_{G} \models \perp$$

$$\Downarrow$$

$$\mathcal{T}_1 \cup G'(\text{pos}) \models \perp$$

$$\Downarrow$$

$$\mathcal{T}_0 \cup G'' \models \perp$$

$$\Phi(c, \overline{c}_{\text{pos}'}, \overline{d}_{\text{pos}}, n, l_{\text{alarm}}, \text{min}, \text{max}, \Delta t) \models \perp$$

Method 1: SAT checking/ Counterexample generation

Method 2: Quantifier elimination

relationships between parameters which guarantee safety

More complex ETCS Case studies

[Faber, Jacobs, VS, 2007]

- Take into account also:
  - Emergency messages
  - Durations

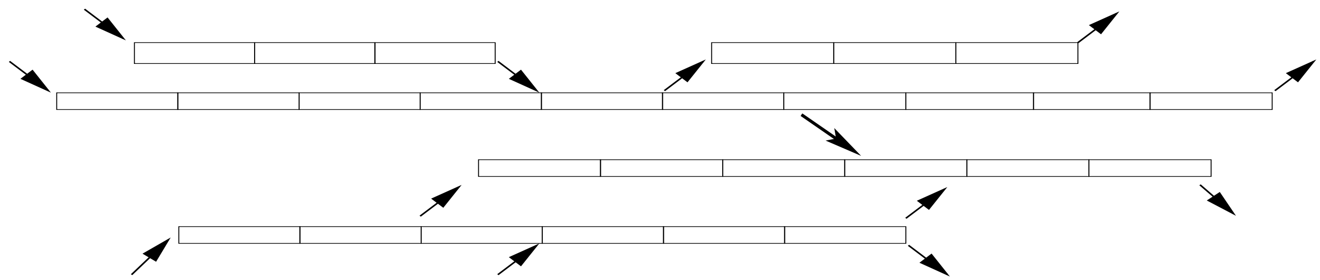
- Specification language: CSP-OZ-DC
  - Reduction to satisfiability in theories for which decision procedures exist

- **Tool chain:** [Faber, Ihlemann, Jacobs, VS]

CSP-OZ-DC $\mapsto$ Transition constr. $\mapsto$ Decision procedures (H-PILoT)

Example 2: Parametric topology

- **Complex track topologies** [Faber, Ihlemann, Jacobs, VS, ongoing work]

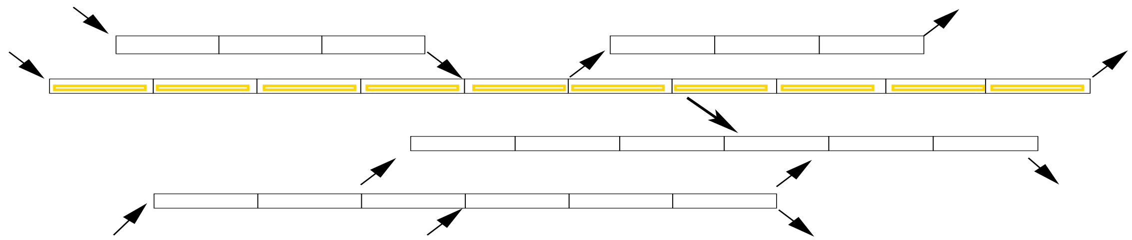


Assumptions:

- No cycles
- in-degree (out-degree) of associated graph at most 2.

Parametricity and modularity

- **Complex track topologies** [Faber, Ihlemann, Jacobs, VS, ongoing work]



Assumptions:

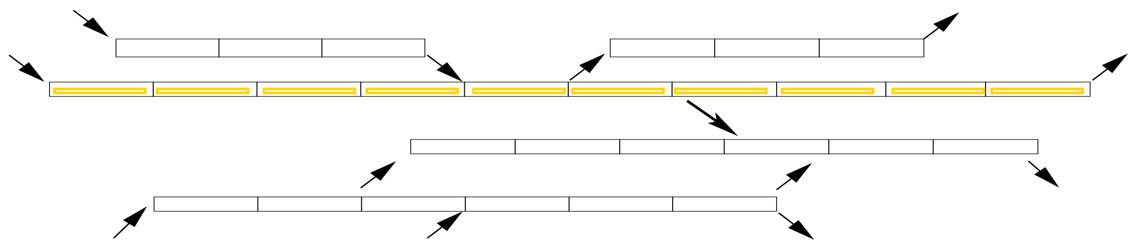
- No cycles
- in-degree (out-degree) of associated graph at most 2.

Approach:

- Decompose the system in trajectories (linear rail tracks; may overlap)
- Task 1: - Prove safety for trajectories with incoming/outgoing trains
  - Conclude that for control rules in which trains have sufficient freedom (and if trains are assigned unique priorities) safety of all trajectories implies safety of the whole system
- Task 2: - General constraints on parameters which guarantee safety

Parametricity and modularity

- **Complex track topologies** [Faber, Ihlemann, Jacobs, VS, ongoing work]



Assumptions:

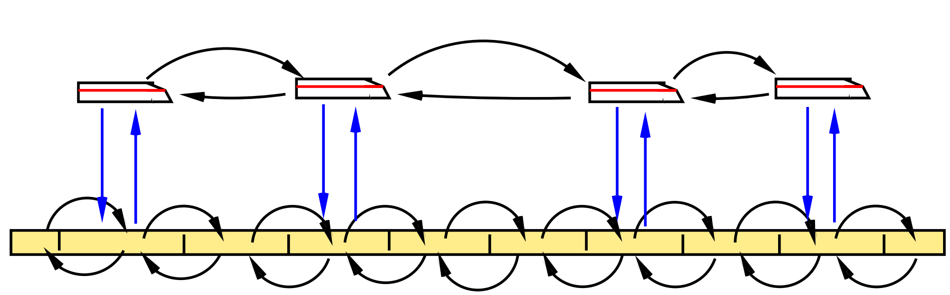
- No cycles
- in-degree (out-degree) of associated graph at most 2.

Data structures:

p_1: trains

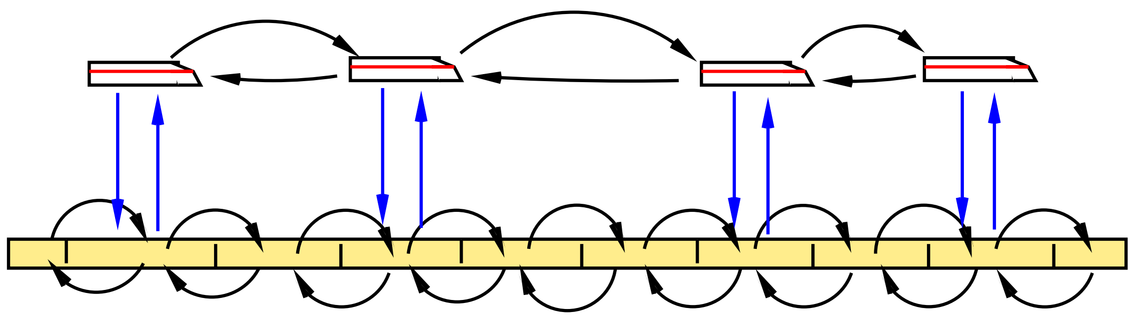
- 2-sorted pointers

p_2: segments



- scalar fields ($f{:}p_i{\rightarrow}\mathbb{R}$, $g{:}p_i{\rightarrow}\mathbb{Z}$)
- updates efficient decision procedures (H-PiLoT)

Incoming and outgoing trains



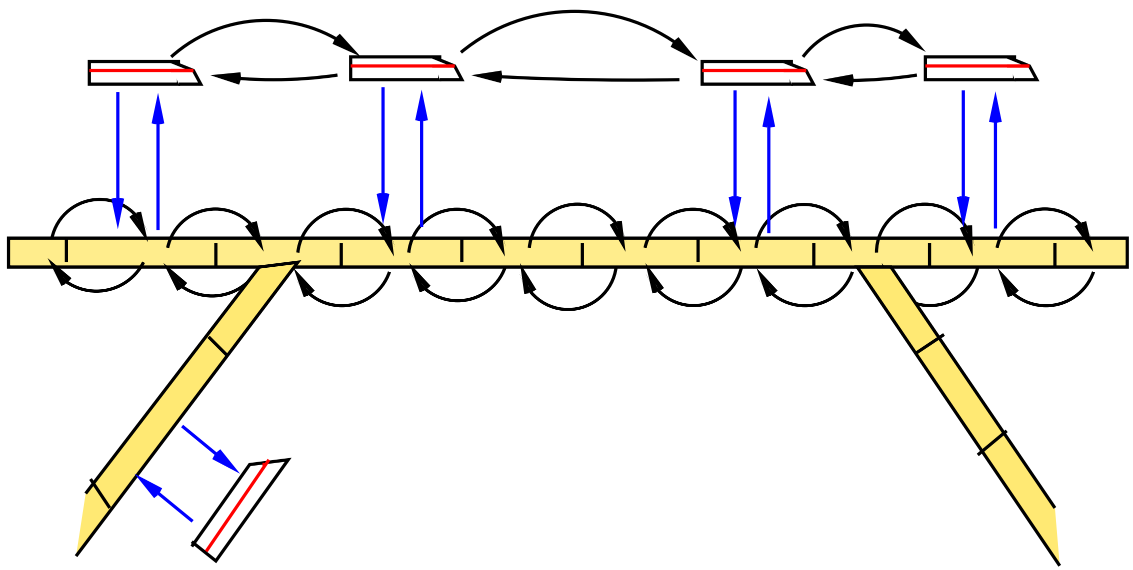
Example 1: Speed Update

$$\mathrm{pos}(t) < \mathrm{length}(\mathrm{segm}(t)) - d \;\rightarrow\; 0 \leq \mathrm{spd}'(t) \leq \mathrm{lmax}(\mathrm{segm}(t))$$

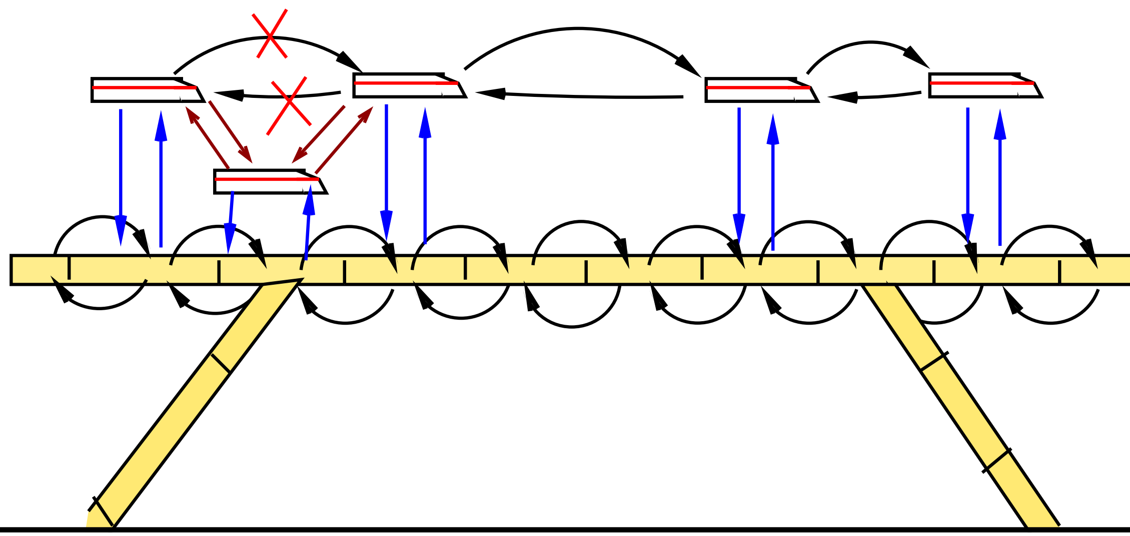
$$\mathrm{pos}(t) \geq \mathrm{length}(\mathrm{segm}(t)) - d \;\wedge\; \mathrm{alloc}(\mathrm{next}_s(\mathrm{segm}(t))) = \mathrm{tid}(t)$$
$$\rightarrow\; 0 \leq \mathrm{spd}'(t) \leq \min(\mathrm{lmax}(\mathrm{segm}(t)), \mathrm{lmax}(\mathrm{next}_s(\mathrm{segm}(t)))$$

$$\mathrm{pos}(t) \geq \mathrm{length}(\mathrm{segm}(t)) - d \;\wedge\; \mathrm{alloc}(\mathrm{next}_s(\mathrm{segm}(t))) \neq \mathrm{tid}(t)$$
$$\rightarrow\; \mathrm{spd}'(t) = \max(\mathrm{spd}(t) - \mathrm{decmax}, 0)$$

Incoming and outgoing trains



Incoming and outgoing trains



Example 2: Enter Update (also updates for segm', spd', pos', train')

Assume: $s_1 \neq \text{null}_s$, $t_1 \neq \text{null}_t$, $\text{train}(s) \neq t_1$, $\text{alloc}(s_1) = \text{idt}(t_1)$

$t \neq t_1, \text{ids}(\text{segm}(t)) < \text{ids}(s_1), \text{next}_t(t) = \text{null}_t, \text{alloc}(s_1) = \text{tid}(t_1) \rightarrow \text{next}'(t) = t_1 \wedge \text{next}'(t_1) = \text{null}_t$

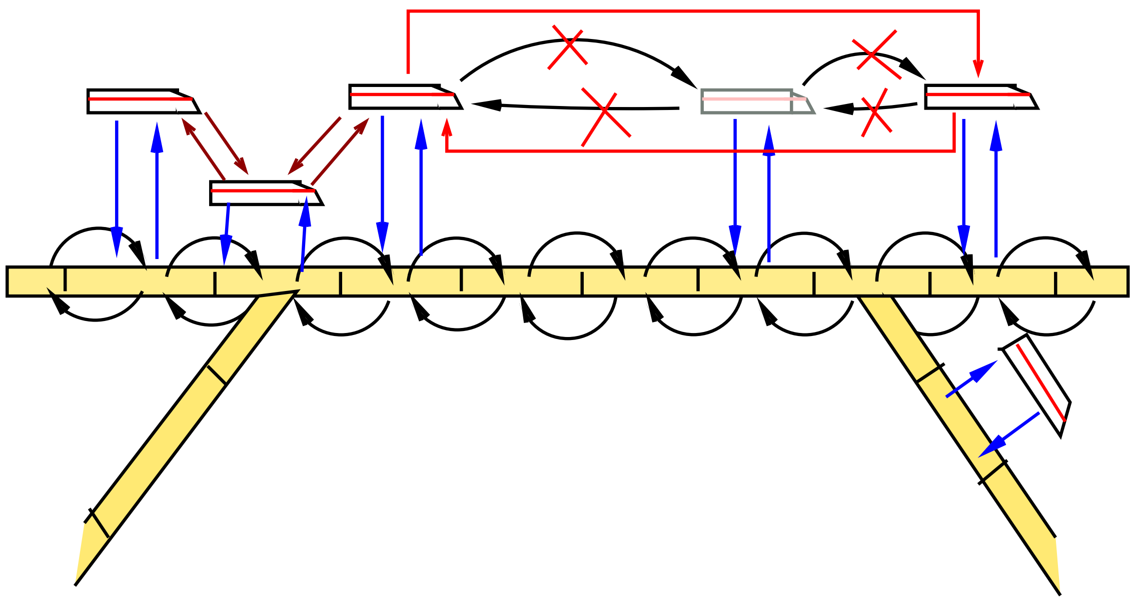
$t \neq t_1, \text{ids}(\text{segm}(t)) < \text{ids}(s_1), \text{alloc}(s_1) = \text{tid}(t_1), \text{next}_t(t) \neq \text{null}_t, \text{ids}(\text{segm}(\text{next}_t(t))) \leq \text{ids}(s_1)$

$\rightarrow \text{next}'(t) = \text{next}_t(t)$

...

$t \neq t_1, \text{ids}(\text{segm}(t)) \geq \text{ids}(s_1) \rightarrow \text{next}'(t) = \text{next}_t(t)$

Incoming and outgoing trains



Safety property

Safety property we want to prove: no two trains ever occupy the same track segment:

$$(\text{Safe}) := \forall t_1, t_2 \;\; \text{segm}(t_1) = \text{segm}(t2) \rightarrow t_1 = t_2$$

In order to prove that (Safe) is an invariant of the system, we need to find a suitable invariant (Inv(i)) for every control location i of the TCS, and prove:

$$(\text{Inv}(i)) \models (\text{Safe}) \text{ for all locations } i$$

and that the invariants are preserved under all transitions of the system,

$$(\text{Inv}(i)) \wedge (\text{Update}) \models (\text{Inv}'(j))$$

whenever (Update) is a transition from location i to j .

Safety property

Need additional invariants.

- generate by hand [Faber, Ihlemann, Jacobs, VS, ongoing]

  use the capabilities of H-PILoT of generating counterexamples

- generate automatically [work in progress]

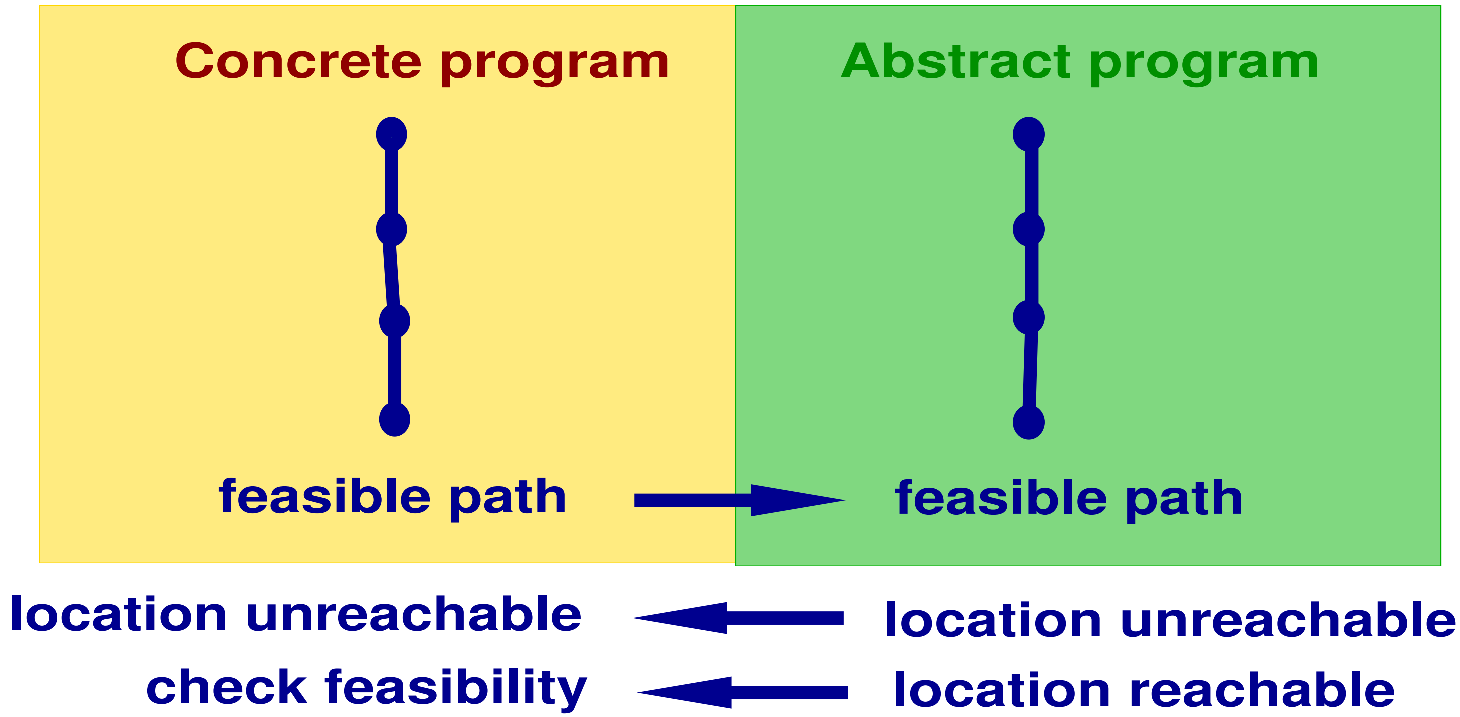
Ground satisfiability problems for pointer data structures

the decision procedures presented before can be used without problems

Other interesting topics

- Generate invariants
- Verification by abstraction/refinement

Abstraction-based Verification



$\Downarrow$

conjunction of constraints: $\phi(1) \wedge Tr(1,2) \wedge \cdots \wedge Tr(n-1,n) \wedge \neg \text{safe}(n)$

- satisfiable: feasible path
- unsatisfiable: refine abstract program s.t. the path is not feasible

[McMillan 2003-2006] use ‘local causes of inconsistency’

$\mapsto$ compute interpolants

Summary

- Decision procedures for various theories/theory combinations

    Implemented in most of the existing SMT provers:
    Z3: `http://z3.codeplex.com/`
    CVC4: `http://cvc4.cs.nyu.edu/web/`
    Yices: `http://yices.csl.sri.com/`

- Ideas about how to use them for verification

Decision procedures for other classes of theories/Applications"
Next semester: Seminar "Decision Procedures and Applications"

More details on Specification, Model Checking, Verification:
This summer (end of August):
Summer school "Verification Technology, Systems & Applications"
Next year: Lecture "Formal Specification and Verification"

Forschungspraktika
BSc/MSc Theses in the area