# Verification of Data Structures Using The Pointer Assertion Logic Engine and Jahob

Feride Çetin

Kremena Diatchka

# Pointer Assertion Logic Engine

- Catch type and memory errors
- Check data structure invariants
- Annotate programs with specifications in Pointer Assertion Logic
- Encode programs in MSOL
- Check the validity using MONA
- Requires *loop and function call invariants*
- Highly modular

# Pointer Assertion Logic

- Store Model
- Graph Types
- The Programming Language
- Program Annotations

# Store Model

- Store ={a heap, program variables}
- Heap = {records}
- Record fields = {pointers, boolean values}
- Pointer value = {null, record}
- Program variables = {data variable, pointer variable}

# Graph Types

- Definition: Tree-shaped data structure with extra pointers.
- Backbone: The underlying tree
- Data fields: Define the backbone
- Pointer fields: Point anywhere in the backbone

# Program Annotations

- PAL: Monadic Second Order Logic on graph types
- Annotations are <u>invariants of the program</u> (formulas) used:
  - To constrain pointer field destinations
  - As loop and procedure call invariants
  - Pre- and post-conditions in procedure declarations
  - in assert and split statements

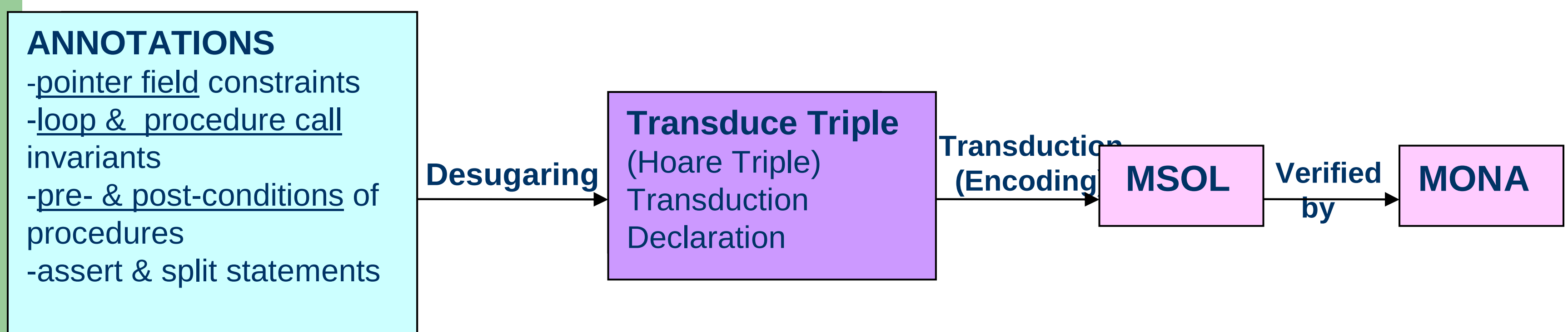
# Pointer fields

- A pointer field must satisfy the formula given in its type declaration unless it is overridden with *pointer directives* of the form:
- *ptrdirs* -> { ( *T* . *p* [ *form* ] )* }
- Allows pointer fields to be constrained differently at different program points.
- Important: Temporary but intentional invalidation of data structure invariants often occurs in imperative programs
- *Well-formed* pointer directives => pointer field denote *exacly one record*

# Properties

- A pair consisting of a formula and a set of pointer directives:
- *property* -> [ *form ptrdirs* ]
- Denotes the set of stores where
  - the formula *form* is satisfied;
  - the data variables denote disjoint acyclic backbones spanning the heap
  - each pointer field satisfies its pointer directive

# Verification pipeline



# Desugaring: Splitting the program into Hoare Triples

- **Modelling transformations of heap with Hoare Triples generated for each cut-point of the program**
- **Form:** *triple -> property stm*
- A triple is valid if
  - executing *stm* in a store where *property* is satisfied cannot violate any assertions occurring in *stm*; and
  - the execution always terminates in a store consisting of disjoint, acyclic backbones spanning the heap in which all pointer directives hold.

# Encoding Hoare Triples

- Encode each Hoare triple in *monadic second-order logic*
  - decidable using MONA
- Transduction technique:
  - Simulate (transduce) the statements
  - Update *store predicates*
  - Check the validity of the resulting formula

# Store predicates

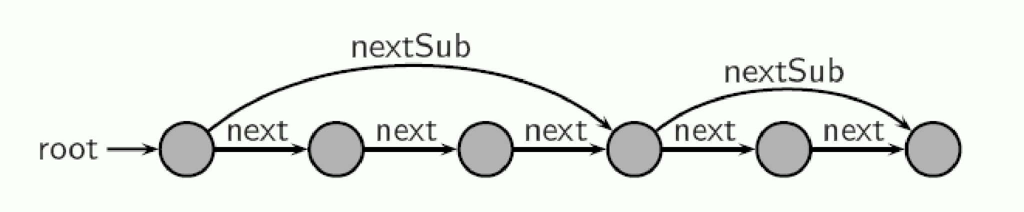
- All properties of a store can be expressed using these predicates in MONA logic
  - bool_*T*_*b*(v)
  - succ_*T*_*d*(v,w)
  - …
- Transduction process -> store predicates for each program point

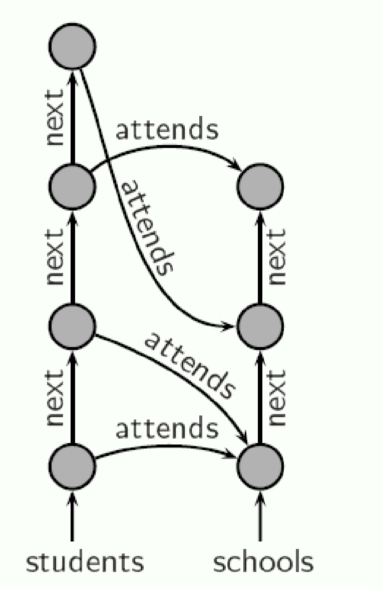
# Summary

- **Pointer Assertion Logic Engine checks:**
  - the pointer directives are well-formed
  - all assertions are valid
  - all cut-point properties are satisfiable
  - memory errors and
  - violations of the data structure invariants
- **Data structures verified using PAL**
  - *Singly-linked lists*
  - *Doubly-linked lists with tail pointers*
  - ...

# PALE limitation

- Problem: all fields not part of the backbone must be exactly determined by the backbone




**2 level skip list**

Backbone field: **`next`**
Derived field: **`nextSub`**

**Students-schools**

Backbone field: **`next`**
Derived field: **`attends`**

# Jahob

- Verifies properties of Java programs with dynamically allocated data structures
- Modular analysis
- Can prove that an implementation
  - satisfies specifications
  - Maintains data structure invariants
  - Never produces run-time errors

# Jahob outline

- Developers give specifications as higher-order logic (HOL) formulas
- Split HOL formulas into conjuncts
- Approximate conjunct
  - Translation to first order logic
  - Field constraint analysis
  - BAPA
- Prove approximation formula using theorem provers and decision procedures

# Field constraint analysis

- Field constraint for a field: formula specifying a set of objects to which the field can point
- Can analyze non-deterministic field constraints
- Approach
  - Verify backbone
  - Verify constraints on cross-cutting fields

# Field Constraint Analysis

- Uniquely determine where fields point to

$$\forall xy. f(x) = y \leftrightarrow F(x, y)$$

- Specify constraint on the field

$$\forall xy. f(x) = y \rightarrow F(x, y)$$

- *f*: function representing the field
- *F*: defining formula for *f*
- Based on approximating *f* with *F*

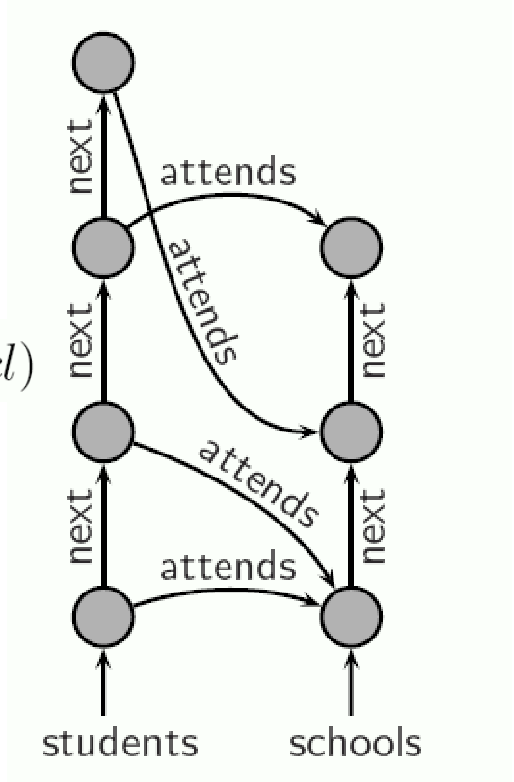
# Students field constraints

$$rtrancl = \{(v, w) \mid v.next = w\}^{*\wedge}$$

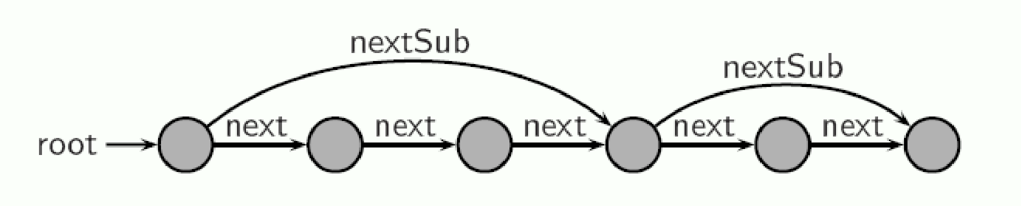
$$a = (students, x) \notin rtrancl) \rightarrow (y = null)$$

$$b = ((students, x) \in rtrancl) \rightarrow ((schools, y) \in rtrancl)$$

$$\forall x\ y\ .\ (x.attends = y) \rightarrow (x \neq null \rightarrow a \wedge b)$$



# Skip list field constraint




$$rtrancl = \{(v, w) \mid v.next = w\}^{*\wedge}$$

$$c = (x = null) \rightarrow (y = null)$$

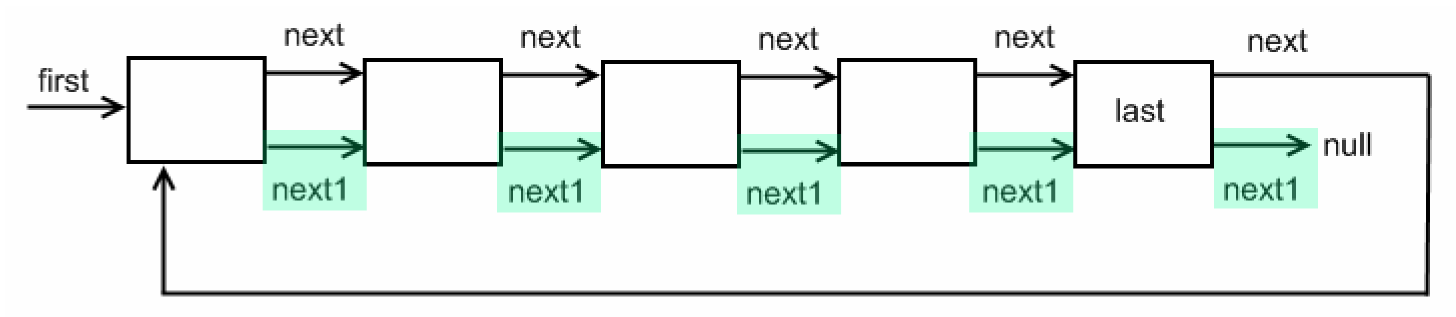
$$d = (x \neq null) \rightarrow ((x.next, y) \in rtrancl)$$

$$\forall x\ y\ .\ (x.nextSub = y) \rightarrow c \wedge d$$

# Project: verifying simple data structures using Jahob

- Verified
  - List with header node
  - Queue
  - Cyclic list
- Tried
  - Instantiable queue
- In progress
  - Leaf-linked tree

# Verifying a cyclic list




- Backbone field: **`next1`**
- Derived field: **`next`**
- Field constraint

$$\forall x\ y\ .\ (x.next = y) \rightarrow (last(x) \rightarrow y = null) \wedge (\neg last(x) \rightarrow y = x.next1)$$

# Verifying a cyclic list: class invariants

```
private static ghost specvar next1 :: "obj => obj";

public static specvar content :: objset;
```

$$\{x \mid (x \neq null) \wedge ((first.next1,\, x) \in \{(v, w) \mid v.next1 = w\}^{*\wedge})\}$$

```
public static specvar isolated :: "obj => bool";
```

$$\lambda n\,.\,(n.next1 = null) \wedge (\forall x\,.\,x \neq null \rightarrow x.next1 \neq n)$$

```
public invariant unallocIsolated:
```

$$\forall n\,.\,n \notin Object.alloc \rightarrow isloted(n)$$

# Verifying a cyclic list: class invariants

```
invariant firstIsolated:
```

$$first \neq null \rightarrow \forall n \,.\, n.next1 \neq first$$

```
private static specvar last :: "obj => bool";
```

$$\lambda n \,.\, ((first, n) \in \{(v, w) \mid v.next1 = w\}^{*\wedge}) \wedge (n.next1 = null)$$

```
invariant isTree: "tree [next1]";
```

```
invariant fieldConstraint:
```

$$\forall x\ y \,.\, (x.next = y) \rightarrow (last(x) \rightarrow y = null) \wedge (\neg last(x) \rightarrow y = x.next1)$$

# The end!

- Verifying data structures this way can be frustrating

- Worth it in safety-critical applications