Verification of Pointer Programs

Stefan Rieger

MOVES: Software Modeling and Verification
RWTH Aachen University, Germany

23.09.2009
Pointers are indispensable and omnipresent

- object-oriented programming
- dynamic memory management and data structures
- data bases and index structures
- ...

Stefan Rieger  Verification of Pointer Programs
Pointers are indispensable and omnipresent

- object-oriented programming
- dynamic memory management and data structures
- data bases and index structures
- ...

Difficulties

- aliasing creates dependencies
- destructive updates
- dereferencing invalid/null pointers

⇒ automatic verification desirable
Example: The Deutsch-Schorr-Waite Algorithm

1. \textbf{if} root = null \textbf{goto} 15;
2. \textbf{new}(sen);
3. prev := sen;
4. cur := root;
5. next := cur.\textit{l};
6. cur.l := cur.r; ←
7. cur.r := prev;
8. prev := cur;
9. cur := next;
10. \textbf{if} (cur = sen) \textbf{goto} 15;
11. \textbf{if} (cur \neq \text{null}) \textbf{goto} 5;
12. cur := prev;
13. prev := null;
14. \textbf{goto} 5;
Example: The Deutsch-Schorr-Waite Algorithm

1. if root = null goto 15;
2. new(sen);
3. prev := sen;
4. cur := root;
5. next := cur.l;
6. cur.l := cur.r;
7. cur.r := prev; ←
8. prev := cur;
9. cur := next;
10. if (cur = sen) goto 15;
11. if (cur \neq null) goto 5;
12. cur := prev;
13. prev := null;
14. goto 5;
Example: The Deutsch-Schorr-Waite Algorithm

```plaintext
1. if root = null goto 15;
2. new(sen);
3. prev := sen;
4. cur := root;
5. next := cur.l;
6. cur.l := cur.r;
7. cur.r := prev;
8. prev := cur; ←
9. cur := next;
10. if (cur = sen) goto 15;
11. if (cur ≠ null) goto 5;
12. cur := prev;
13. prev := null;
14. goto 5;
```
Example: The Deutsch-Schorr-Waite Algorithm

```plaintext
1 if root = null goto 15;
2 new(sen);
3 prev := sen;
4 cur := root;
5 next := cur.l;
6 cur.l := cur.r; ←
7 cur.r := prev;
8 prev := cur;
9 cur := next;
10 if (cur = sen) goto 15;
11 if (cur ≠ null) goto 5;
12 cur := prev;
13 prev := null;
14 goto 5;
```
Example: The Deutsch-Schorr-Waite Algorithm

1. if root = null goto 15;
2. new(sen);
3. prev := sen;
4. cur := root;
5. next := cur.l;
6. cur.l := cur.r;
7. cur.r := prev;
8. prev := cur; ←
9. cur := next;
10. if (cur = sen) goto 15;
11. if (cur ≠ null) goto 5;
12. cur := prev;
13. prev := null;
14. goto 5;
Example: The Deutsch-Schorr-Waite Algorithm

1. if root = null goto 15;
2. new(sen);
3. prev := sen;
4. cur := root;
5. next := cur.l;
6. cur.l := cur.r; ←
7. cur.r := prev;
8. prev := cur;
9. cur := next;
10. if (cur = sen) goto 15;
11. if (cur ≠ null) goto 5;
12. cur := prev;
13. prev := null;
14. goto 5;
Example: The Deutsch-Schorr-Waite Algorithm

1. if root = null goto 15;
2. new(sen);
3. prev := sen;
4. cur := root;
5. next := cur.l;
6. cur.l := cur.r; ←
7. cur.r := prev;
8. prev := cur;
9. cur := next;
10. if (cur = sen) goto 15;
11. if (cur ≠ null) goto 5;
12. cur := prev;
13. prev := null;
14. goto 5;
Example: The Deutsch-Schorr-Waite Algorithm

```
1 if root = null goto 15;
2 new(sen);
3 prev := sen;
4 cur := root;
5 next := cur.l;
6 cur.l := cur.r;
7 cur.r := prev; ←
8 prev := cur;
9 cur := next;
10 if (cur = sen) goto 15;
11 if (cur ≠ null) goto 5;
12 cur := prev;
13 prev := null;
14 goto 5;
```
Example: The Deutsch-Schorr-Waite Algorithm

1. if root = null goto 15;
2. new(sen);
3. prev := sen;
4. cur := root;
5. next := cur.l;
6. cur.l := cur.r;
7. cur.r := prev;
8. prev := cur; ←
9. cur := next;
10. if (cur = sen) goto 15;
11. if (cur ≠ null) goto 5;
12. cur := prev;
13. prev := null;
14. goto 5;
**Example: The Deutsch-Schorr-Waite Algorithm**

1. \textbf{if} root = null \textbf{goto} 15;
2. \textbf{new}(sen);
3. prev := sen;
4. cur := root;
5. next := cur.\textit{l};
6. cur.\textit{l} := cur.\textit{r}; ←
7. cur.\textit{r} := prev;
8. prev := cur;
9. cur := next;
10. \textbf{if} (cur = sen) \textbf{goto} 15;
11. \textbf{if} (cur \neq \text{null}) \textbf{goto} 5;
12. cur := prev;
13. prev := \text{null};
14. \textbf{goto} 5;
Example: The Deutsch-Schorr-Waite Algorithm

1. if root = null goto 15;
2. new(sen);
3. prev := sen;
4. cur := root;
5. next := cur.l; ←
6. cur.l := cur.r;
7. cur.r := prev;
8. prev := cur;
9. cur := next;
10. if (cur = sen) goto 15;
11. if (cur ≠ null) goto 5;
12. cur := prev;
13. prev := null;
14. goto 5;
Example: The Deutsch-Schorr-Waite Algorithm

1. if root = null goto 15;
2. new(sen);
3. prev := sen;
4. cur := root;
5. next := cur.l;
6. cur.l := cur.r; ←
7. cur.r := prev;
8. prev := cur;
9. cur := next;
10. if (cur = sen) goto 15;
11. if (cur ≠ null) goto 5;
12. cur := prev;
13. prev := null;
14. goto 5;
Example: The Deutsch-Schorr-Waite Algorithm

1. if root = null goto 15;
2. new(sen);
3. prev := sen;
4. cur := root;
5. next := cur.\textit{l};
6. cur.\textit{l} := cur.\textit{r};
7. cur.\textit{r} := prev;
8. prev := cur; ←
9. cur := next;
10. if (cur = sen) goto 15;
11. if (cur \neq \text{null}) goto 5;
12. cur := prev;
13. prev := \text{null};
14. goto 5;
Example: The Deutsch-Schorr-Waite Algorithm

1. if root = null goto 15;
2. new(sen);
3. prev := sen;
4. cur := root;
5. next := cur.l; ←
6. cur.l := cur.r;
7. cur.r := prev;
8. prev := cur;
9. cur := next;
10. if (cur = sen) goto 15;
11. if (cur ≠ null) goto 5;
12. cur := prev;
13. prev := null;
14. goto 5;
Example: The Deutsch-Schorr-Waite Algorithm

1. if root = null goto 15;
2. new(sen);
3. prev := sen;
4. cur := root;
5. next := cur.l;
6. cur.l := cur.r; ←
7. cur.r := prev;
8. prev := cur;
9. cur := next;
10. if (cur = sen) goto 15;
11. if (cur \neq null) goto 5;
12. cur := prev;
13. prev := null;
14. goto 5;
Example: The Deutsch-Schorr-Waite Algorithm

```plaintext
1 if root = null goto 15;
2 new(sen);
3 prev := sen;
4 cur := root;
5 next := cur.l;
6 cur.l := cur.r;
7 cur.r := prev; ←
8 prev := cur;
9 cur := next;
10 if (cur = sen) goto 15;
11 if (cur ≠ null) goto 5;
12 cur := prev;
13 prev := null;
14 goto 5;
```
Example: The Deutsch-Schorr-Waite Algorithm

1 \textbf{if} root = null \textbf{goto} 15;  
2 new(sen);  
3 prev := sen;  
4 cur := root;  
5 next := cur.l;  
6 cur.l := cur.r;  
7 cur.r := prev;  
8 prev := cur;  
9 cur := next;  
10 \textbf{if} (cur = sen) \textbf{goto} 15;  
11 \textbf{if} (cur \neq \text{null}) \textbf{goto} 5;  
12 cur := prev;  
13 prev := null;  
14 \textbf{goto} 5;  

\begin{tikzpicture}
  \node (root) at (0,0) {$\text{root}$};
  \node (v) at (0,-2) {$v$};
  \node (l) at (-1,-4) {$l$};
  \node (r) at (1,-4) {$r$};
  \node (prev) at (-2,-6) {$\text{prev}$};
  \node (next) at (2,-6) {$\text{next}$};
  \node (S) at (-2,-8) {$S$};
  \node (U) at (2,-8) {$U$};
  \draw (root) -- (v);
  \draw (v) -- (l);
  \draw (v) -- (r);
  \draw (v) -- (prev);
  \draw (v) -- (next);
\end{tikzpicture}
Example: The Deutsch-Schorr-Waite Algorithm

1. if root = null goto 15;
2. new(sen);
3. prev := sen;
4. cur := root;
5. next := cur.l;
6. cur.l := cur.r;
7. cur.r := prev;
8. prev := cur;
9. cur := next;
10. if (cur = sen) goto 15;
11. if (cur ≠ null) goto 5;
12. cur := prev;
13. prev := null;
14. goto 5;
Example: The Deutsch-Schorr-Waite Algorithm

1. if root = null goto 15;
2. new(sen);
3. prev := sen;
4. cur := root;
5. next := cur.l;
6. cur.l := cur.r;
7. cur.r := prev;
8. prev := cur;
9. cur := next;
10. if (cur = sen) goto 15;
11. if (cur ≠ null) goto 5;
12. cur := prev;
13. prev := null;
14. goto 5;
Example: The Deutsch-Schorr-Waite Algorithm

1. if root = null goto 15;
2. new(sen);
3. prev := sen;
4. cur := root;
5. next := cur.l;
6. cur.l := cur.r; ←
7. cur.r := prev;
8. prev := cur;
9. cur := next;
10. if (cur = sen) goto 15;
11. if (cur ≠ null) goto 5;
12. cur := prev;
13. prev := null;
14. goto 5;
Example: The Deutsch-Schorr-Waite Algorithm

1. if root = null goto 15;
2. new(sen);
3. prev := sen;
4. cur := root;
5. next := cur.l;
6. cur.l := cur.r; ←
7. cur.r := prev;
8. prev := cur;
9. cur := next;
10. if (cur = sen) goto 15;
11. if (cur ≠ null) goto 5;
12. cur := prev;
13. prev := null;
14. goto 5;
Example: The Deutsch-Schorr-Waite Algorithm

```plaintext
1 if root = null goto 15;
2 new(sen);
3 prev := sen;
4 cur := root;
5 next := cur.l;
6 cur.l := cur.r;
7 cur.r := prev; ←
8 prev := cur;
9 cur := next;
10 if (cur = sen) goto 15;
11 if (cur != null) goto 5;
12 cur := prev;
13 prev := null;
14 goto 5;
```
Example: The Deutsch-Schorr-Waite Algorithm

1. if root = null goto 15;
2. new(sen);
3. prev := sen;
4. cur := root;
5. next := cur.l;
6. cur.l := cur.r;
7. cur.r := prev;
8. prev := cur;  ←
9. cur := next;
10. if (cur = sen) goto 15;
11. if (cur ≠ null) goto 5;
12. cur := prev;
13. prev := null;
14. goto 5;
Example: The Deutsch-Schorr-Waite Algorithm

1. if root = null goto 15;
2. new(sen);
3. prev := sen;
4. cur := root;
5. next := cur.l;
6. cur.l := cur.r; ←
7. cur.r := prev;
8. prev := cur;
9. cur := next;
10. if (cur = sen) goto 15;
11. if (cur ≠ null) goto 5;
12. cur := prev;
13. prev := null;
14. goto 5;
Example: The Deutsch-Schorr-Waite Algorithm

1 if root = null goto 15;
2 new(sen);
3 prev := sen;
4 cur := root;
5 next := cur.l;
6 cur.l := cur.r; ←
7 cur.r := prev;
8 prev := cur;
9 cur := next;
10 if (cur = sen) goto 15;
11 if (cur ≠ null) goto 5;
12 cur := prev;
13 prev := null;
14 goto 5;
Example: The Deutsch-Schorr-Waite Algorithm

1. \textbf{if} root = null \textbf{goto} 15;
2. \textbf{new}(sen);
3. prev := sen;
4. cur := root;
5. next := cur.l;
6. cur.l := cur.r;
7. cur.r := prev;
8. prev := cur; ←
9. cur := next;
10. \textbf{if} (cur = sen) \textbf{goto} 15;
11. \textbf{if} (cur \neq \text{null}) \textbf{goto} 5;
12. cur := prev;
13. prev := \text{null};
14. \textbf{goto} 5;
Example: The Deutsch-Schorr-Waite Algorithm

1. if root = null goto 15;
2. new(sen);
3. prev := sen;
4. cur := root;
5. next := cur.
l;
6. cur.
l := cur.
r;
7. cur.
r := prev;
8. prev := cur;
9. cur := next;
10. if (cur = sen) goto 15;
11. if (cur ≠ null) goto 5;
12. cur := prev;
13. prev := null;
14. goto 5;
Example: The Deutsch-Schorr-Waite Algorithm

1. \textbf{if} \ root = \text{null} \textbf{goto} \ 15;
2. \textbf{new}(\text{sen});
3. \text{prev} := \text{sen};
4. \text{cur} := \text{root};
5. \text{next} := \text{cur}.l;
6. \text{cur}.l := \text{cur}.r;
7. \text{cur}.r := \text{prev};
8. \text{prev} := \text{cur};
9. \text{cur} := \text{next};
10. \textbf{if} \ (\text{cur} = \text{sen}) \textbf{goto} \ 15;
11. \textbf{if} \ (\text{cur} \neq \text{null}) \textbf{goto} \ 5;
12. \text{cur} := \text{prev};
13. \text{prev} := \text{null};
14. \textbf{goto} \ 5;
Example: The Deutsch-Schorr-Waite Algorithm

1. if root = null goto 15;
2. new(sen);
3. prev := sen;
4. cur := root;
5. next := cur.l;
6. cur.l := cur.r;
7. cur.r := prev;
8. prev := cur; ←
9. cur := next;
10. if (cur = sen) goto 15;
11. if (cur ≠ null) goto 5;
12. cur := prev;
13. prev := null;
14. goto 5;
Verification of Pointer Structures

<table>
<thead>
<tr>
<th>Problems</th>
</tr>
</thead>
<tbody>
<tr>
<td>• handling inputs of arbitrary size</td>
</tr>
<tr>
<td>• dynamic memory allocation at runtime</td>
</tr>
<tr>
<td>⇒ possibly infinite state space</td>
</tr>
</tbody>
</table>
## Verification of Pointer Structures

### Problems
- handling inputs of arbitrary size
- dynamic memory allocation at runtime
  ⇒ possibly infinite state space

### Approach: Over-Approximation by Abstraction
- use HRGs to model data structures
- abstraction and concretization based on HRG rules
  ⇒ finite state spaces for e.g. model checking
Verification of Pointer Structures

Problems

- handling inputs of arbitrary size
- dynamic memory allocation at runtime
  ⇒ possibly infinite state space

Approach: Over-Approximation by Abstraction

- use HRGs to model data structures
- abstraction and concretization based on HRG rules
  ⇒ finite state spaces for e.g. model checking

Simple Pointer Programming Language (only pointers as data)

- pointer assignment ($x.a := y.b$)
- creation of objects ($\text{new}(x)$)
- limited dereferencing depth (no real restriction)
**Related Work**

**Shape Analysis** represents unbounded heap graphs by three-valued logical structures [Sagiv et al., 2002, Beyer et al., 2006]

**Separation Logic** is an extension of Hoare logic [Reynolds, 2002, O’Hearn et al., 2004]

**Graph Transformation** is used in different approaches:
- abstraction and verification of graph transformation systems [Baldan and König, 2002, Baldan et al., 2004, Kastenberg and Rensink, 2006]
- model pointer assignments directly by graph transformations [Rensink, 2004, Rensink and Distefano, 2006]
- graph reduction grammars [Bakewell et al., 2004a, Bakewell et al., 2004b, Dodds and Plump, 2006]
Alphabets and Hypergraphs

Ranked Alphabet $\Sigma$
- ranking function $rk : \Sigma \rightarrow \mathbb{N}$
- $\Sigma$ consists of terminals and nonterminals: $\Sigma = T_\Sigma \cup N_\Sigma$

Hypergraphs
- hyperedges connect an arbitrary number of vertices
- hyperedges are labeled with symbols from $\Sigma$
- rank of label determines the arity of the edge
- external vertices are used for hyperedge replacements
Representing Heap States

<table>
<thead>
<tr>
<th>Heapgraph → Hypergraph</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Rank of Edges</strong></td>
</tr>
<tr>
<td>pointers</td>
</tr>
<tr>
<td>program variables</td>
</tr>
<tr>
<td>abstract subgraphs</td>
</tr>
</tbody>
</table>

- **Heapgraph**
  - **Pointers**: `P`, `l`, `r` (with selector `s`)
  - **Program Variables**: `x`, `X` (terminal)
  - **Nonterminal Edge**: `T`
  - **Nonterminal**: `T`

- **Hypergraph**
  - **Pointers**: `P`, `l`, `r` (with selector `s`)
  - **Program Variables**: `x`, `X` (terminal)
  - **Nonterminal Edge**: `T`
  - **Nonterminal**: `T`

- **Diagram Notes**
  - Omit tentacle numbers when order clear

Stefan Rieger  Verification of Pointer Programs
Concrete and Abstract Heaps

Abstract Heap

A heap configuration (=hypergraph) is abstract, if it contains at least one nonterminal edge.
Concrete and Abstract Heaps

Abstract Heap

A heap configuration (=hypergraph) is abstract, if it contains at least one nonterminal edge.

Admissibility

A heap configuration is admissible if nodes referred by variables are not adjacent to nonterminal edges.

Inadmissible

Useful for abstract semantics ("concrete assignment").
Overview

1. Hyperedge Replacement
2. Abstraction and Concretization
3. Pointer Logic
4. Verification and Model Checking
Executing a hyperedge replacement

1. Hypergraph \( H \) with hyperedge \( e \in E_H \) s.t. \( \ell(e) \in N_\Sigma \)

2. Hypergraph \( R \) with \( |ext_R| = rk(e) \)

Example
Hyperedge Replacement Grammars

Definition

A HRG $G$ is a set of productions of the form $X \rightarrow R$ with $X \in N_{\Sigma}$ and hypergraph $R$ where $|\text{ext}_R| = rk(X)$.

Example: HRG for (fully branched) Binary Trees

```
      1
     / \  \
    l   r  \
   / \    \
  T   T  T
```

```
      1
     / \  \
    l   r  \
   / \    \
  T   T  T
```
Properties

**Context-freeness**
HRGs are context-free and confluent.

**Applicability**
A rule is applicable to a hypergraph if it contains a nonterminal that matches the rule’s LHS.

**Derivation**
A derivation is a sequence \( H_0 \xrightarrow{G} H_1 \xrightarrow{G} H_2 \xrightarrow{G} \ldots \) where each \( H_i \xrightarrow{G} H_{i+1} \) is a application of a rule from \( G \).
Properties

Context-freeness
HRGs are context-free and confluent.

Applicability
A rule is applicable to a hypergraph if it contains a nonterminal that matches the rule’s LHS.

Derivation
A derivation is a sequence $H_0 \xrightarrow{G} H_1 \xrightarrow{G} H_2 \xrightarrow{G} \ldots$ where each $H_i \xrightarrow{G} H_{i+1}$ is a application of a rule from $G$.

Graph Language of HRG $G$

$$\mathcal{L}(G, H) = \{ K \in \text{HGraph}_{T_\Sigma} \mid H \xrightarrow{G}^* K \}$$

(= all terminal graphs which are derivable from $H$)
Overview

1. Hyperedge Replacement

2. Abstraction and Concretization

3. Pointer Logic

4. Verification and Model Checking
Abstracting the Heap

**Abstraction**

For HRG $G$ and hypergraph $H$ the set of abstractions of $H$ is

$$\text{Abstractions}(H) = \{ K \in \text{HGraph}_\Sigma \mid K \xrightarrow{G}^+ H \}$$

If LHS $<\text{RHS}$ for all rules in $G$, $\text{Abstractions}(H)$ is finite.

**Idea**

- Compute abstractions by reverse application of HRG rules
- Reverse application requires finding a subgraph isomorphism
- Reverse rule application is not confluent
Abstraction Example - Binary Trees

\( T \rightarrow \)

\[
\begin{array}{c}
\text{left} \quad \text{right} \\
T & T \\
T & T \\
T & T \\
T & T \\
\end{array}
\]
Path Abstraction
Correctness (but Over-Approximation)

By definition every concrete heap configuration can be regenerated from its abstractions.

\[
\text{Abstractions}(H) = \{ K \in \text{HGraph}_\Sigma \mid K \stackrel{g}{\longrightarrow}^+ H \}\]
Abstracting the Heap II

Correctness (but Over-Approximation)

By definition every concrete heap configuration can be regenerated from its abstractions.

\[
\text{Abstractions}(H) = \{K \in \text{HGraph}_\Sigma \mid K \xrightarrow{g}^+ H\}
\]

Abstraction alone insufficient

- Assignment easy since admissibility guarantees concrete edges near variables.
- **But:** assignments may yield inadmissible configurations
- **Idea:** materialize concrete objects from nonterminals (partial concretization)
Partial Concretization by Forward Rule Application

\[ P \rightarrow \]

\[ \begin{align*}
    & 1 & & 1 \\
    & l & & l \\
    & P & & P \\
    & 2 & & 2 \\
\end{align*} \]

\[ \begin{align*}
    & 1 & & 1 \\
    & r & & r \\
    & P & & P \\
    & 2 & & 2 \\
\end{align*} \]
Resulting Hypergraphs
Different Situation

Diagram showing a process or transformation involving pointers and nodes, with labels and structures indicating movement or interaction.
Inadmissible Results
The Solution

Heap Abstraction Grammars

- introduce **redundant** rules allowing concretization “from below”
- additional rules must guarantee **completeness**

Additional Rules

```
1  1
P  l r
2  2
1  1
P  l r
2  2
1  2
P
2
```

```
1  1
P  l r
2  2
1  1
P  l r
2  2
1  1
P
2
```
Overview

1. Hyperedge Replacement
2. Abstraction and Concretization
3. Pointer Logic
4. Verification and Model Checking
Temporal Pointer Logic

- Combination of LTL operators and pointer comparisons
- Arbitrarily deep dereferencing

Formal Definition

Let $\mathcal{F}$ be a set of flags ($\text{err, term, ...}$).

$$\text{TPL}(\Sigma, \mathcal{F}) ::= \text{true} \mid \mathcal{F} \mid \text{DEREF}_\Sigma = \text{DEREF}_\Sigma$$

$$\mid \neg \text{TPL}(\Sigma, \mathcal{F}) \mid \text{TPL}(\Sigma, \mathcal{F}) \land \text{TPL}(\Sigma, \mathcal{F})$$

$$\mid \mathbf{X} \text{TPL}(\Sigma, \mathcal{F}) \mid \text{TPL}(\Sigma, \mathcal{F}) \mathbf{U} \text{TPL}(\Sigma, \mathcal{F})$$

$$\text{DEREF}_\Sigma ::= \text{null} \mid \text{Var}_\Sigma \mid \text{DEREF}_\Sigma \cdot \text{Sel}_\Sigma$$

$$\text{F}\varphi = \text{true} \ \text{U} \ \varphi \quad \text{G} \ \varphi = \neg \text{F} \ \neg\varphi$$
Semantics of TPL

Interpretation

- Interpret TPL formulae on infinite and finite sequences of heap configurations.
- Every trace of heap configurations has an associated trace of (sets of) flags of equal length.

Finite Traces

Let $t \in \text{aHHC}_\Sigma^*$ and $u \in \mathcal{F}^*$ be a finite traces of length $n$. Implicit extension as follows:

$t(1) \ t(2) \ \ldots \ \ t(n) \ \ t(n) \ \ t(n) \ \ \ldots$

$u(1) \ u(2) \ \ldots \ u(n) \ u(n) \cup \{\text{term}\} \ u(n) \cup \{\text{term}\} \ \ldots$
Formal Semantics of Pointer Comparisons

Concrete Semantics

$$\text{CSat}[\xi = \zeta, H] = \begin{cases} 
1 & \text{if } D[\xi, H] = D[\zeta, H] \neq \bot \\
0 & \text{otherwise}
\end{cases}$$

$$D[\zeta, H]$$: “intuitive” concrete expression semantics
Formal Semantics of Pointer Comparisons

Concrete Semantics

\[ CSAT[\xi = \zeta, H] = \begin{cases} 
1 & \text{if } D[\xi, H] = D[\zeta, H] \neq \perp \\
0 & \text{otherwise}
\end{cases} \]

\( D[\zeta, H] \): “intuitive” concrete expression semantics

Abstract Semantics – 3 cases

\[ ASAT[\gamma, H] = \begin{cases} 
1 & \text{if } \forall H' \in \mathcal{L}(G, H) : CSAT[\gamma, H'] = 1 \\
0 & \text{if } \forall H' \in \mathcal{L}(G, H) : CSAT[\gamma, H'] = 0 \\
\frac{1}{2} & \text{otherwise}
\end{cases} \]
Overview

1. Hyperedge Replacement
2. Abstraction and Concretization
3. Pointer Logic
4. Verification and Model Checking
Hyperedge Replacement

Abstraction and Concretization

Pointer Logic

Verification and Model Checking

**Pointer Program**

**HRG**

**TPL Formula**

**Abstract Transition System**

**Pointer Comparisons**

**Evaluation of Pointer Comparisons on Abstract States**

**Partial Transition System** *(labeled by 3-valued atomic propositions)*

**Interpret TPL Formula as LTL Formula**

**Completion of Partial Transition System**

**3-valued LTL Model Checking**

**Standard LTL Model Checking**
Evaluating Pointer Comparisons

$x.a_1.a_2...a_m = y.b_1.b_2...b_n$

- Use two auxiliary variables $t_1$ and $t_2$ to walk along “paths”
- Assignments followed (not preceeded) by concretization steps
- Check if in all concretizations $t_1 = t_2$
Does $\text{ASAT}[x.r.l.r = y.l.r, H] = 1$ hold?
Does $\text{ASAT}[x.r.l.r = y.l.r, H] = 1$ hold?
Does \( \text{ASAT}[x.r.l.r = y.l.r, H] = 1 \) hold?
Does $\text{ASAT} \left[ x.r.l.r = y.l.r, H \right] = 1$ hold?
Does $\text{ASAT} \left[ x.r.l.r = y.l.r, H \right] = 1$ hold?
Does $\text{ASAT}[x.r.l.r = y.l.r, H] = 1$ hold?
Does $\text{ASAT}[x.r.l.r = y.l.r, H] = 1$ hold?
\( \forall K : \text{CSAT}[t_1 = t_2, H] = 1 \implies \text{ASAT}[...] = 1 \)
A Special Case

Limiting Dereferencing Depth

When dereferencing depth in pointer comparisons is limited to one, we always get clearly determined results (0 or 1).

Reason: admissibility of heap configurations
Hyperedge Replacement
Abstraction and Concretization
Pointer Logic
Verification and Model Checking

- Pointer Program
- HRG
- TPL Formula
  - Abstract Transition System
  - Pointer Comparisons
  - Evaluation of Pointer Comparisons on Abstract States
  - Partial Transition System (labeled by 3-valued atomic propositions)
  - Interpret TPL Formula as LTL Formula
    - 3-valued LTL Model Checking
    - Completion of Partial Transition System
      - Standard LTL Model Checking
### Three-valued LTL Model Checking

#### Setting
- Evaluation of pointer comparisons can result in either 0, 1 or $\frac{1}{2}$
- Transition system has 3-valued labeling

#### Transformation

Transform transition system to represent all possibilities for $\frac{1}{2}$-valued predicates.
Hyperedge Replacement

Abstraction and Concretization

Pointer Logic

Verification and Model Checking

Pointer Program → HRG → Abstract Transition System → Evaluation of Pointer Comparisons on Abstract States → Partial Transition System (labeled by 3-valued atomic propositions) → Interpret TPL Formula as LTL Formula → 3-valued LTL Model Checking

Completion of Partial Transition System → Standard LTL Model Checking

Stefan Rieger Verification of Pointer Programs
Quantifiers

Quantified TPL

\[ Q_1 X_1 Q_2 X_2 \ldots Q_k X_k : \varphi(X_1, X_2, \ldots, X_k) \]

- Quantification over heap objects present in the initial states
- Preservation of object identities between states by nondeterministic marking with variables
- For every quantor an additional marking is necessary (exponential blow-up of state space)
Example: The Deutsch-Schorr-Waite Algorithm

**Pointer Safety:** No pointer errors / null dereferences

**Shape Safety:** Input structure is retained

**Completeness:** all vertices are visited at least once

\[ \forall X : \neg (\text{cur} \neq X \cup \text{term}) \]

**Termination:** finally \( X \) never points to \( \text{cur} \) anymore

\[ \forall X : \text{FG}(\text{cur} \neq X) \]

**Correctness:** for all vertices the left- and right successors are the same after program termination

\[ \forall X \forall X_l \forall X_r : X.l = X_l \land X.r = X_r \rightarrow \\
((X = \text{root} \rightarrow \text{G}(X = \text{root})) \\
\land \text{G}(\text{term} \rightarrow (X.l = X_l \land X.r = X_r))) \]
Experimental Results: Verifying the DSW Algorithm

- An initial heap:
  - `root`
  - `l` -> `T`
  - `r` -> `T`

- A final heap:
  - `root`
  - `prev` -> `l` -> `T`
  - `cur` -> `T`
  - `next`

- An intermediate heap state:
  - `root`
  - `prev` -> `l` -> `T`
  - `cur`
  - `next`
## Experimental Results: Verifying the DSW Algorithm

<table>
<thead>
<tr>
<th></th>
<th>no marking</th>
<th>1 marking</th>
<th>3 markings</th>
<th>TVLA</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Initial States</strong></td>
<td>5</td>
<td>185</td>
<td>962</td>
<td></td>
</tr>
<tr>
<td><strong>Number of States</strong></td>
<td>20,678</td>
<td>6,220,798</td>
<td>35,983,627</td>
<td>&gt; 80,000</td>
</tr>
<tr>
<td><strong>Number of Transitions</strong></td>
<td>23,359</td>
<td>7,078,257</td>
<td>40,909,648</td>
<td></td>
</tr>
<tr>
<td><strong>State Space Gen. (h:min:sec)</strong></td>
<td>&lt;0:01</td>
<td>10:14</td>
<td>1:18:03</td>
<td></td>
</tr>
<tr>
<td><strong>Memory Consumption</strong></td>
<td>41 MB</td>
<td>788 MB</td>
<td>3,900 MB</td>
<td>150 MB</td>
</tr>
<tr>
<td><strong>Pointer Safety</strong></td>
<td>on-the-fly</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td><strong>Shape Safety</strong></td>
<td>on-the-fly</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td><strong>Completeness (min:sec)</strong></td>
<td>-</td>
<td>0:16</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td><strong>Termination (min:sec)</strong></td>
<td>-</td>
<td>0:39</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td><strong>Correctness (min:sec)</strong></td>
<td>-</td>
<td>-</td>
<td>4:05</td>
<td></td>
</tr>
<tr>
<td><strong>Total Time (State Space Gen. + all Properties)</strong></td>
<td>1:28:35</td>
<td>-</td>
<td>&lt;9:00:00</td>
<td></td>
</tr>
</tbody>
</table>
Conclusion

- analysis and verification of complex data structures
- highly parametrized framework
- handling of inconsistencies wrt. the data structure
- more intuitive than other approaches
- promising experimental results
Conclusion

- analysis and verification of complex data structures
- highly parametrized framework
- handling of inconsistencies wrt. the data structure
- more intuitive than other approaches
- promising experimental results

Additional Features

- abstraction-only grammars
- optimized concretization possible
- unbounded thread creation [Noll and Rieger, 2008]
Conclusion

- analysis and verification of complex data structures
- highly parametrized framework
- handling of inconsistencies wrt. the data structure
- more intuitive than other approaches
- promising experimental results

Additional Features

- abstraction-only grammars
- optimized concretization possible
- unbounded thread creation [Noll and Rieger, 2008]

Outlook

- learning of HRGs
- typed/attributed HRGs
(FM 2008) Thomas Noll and Stefan Rieger. Verifying Dynamic Pointer-Manipulating Threads

(ICGT 2008) Stefan Rieger and Thomas Noll. Abstracting Complex Data Structures by Hyperedge Replacement


(ICTAC 2007) Thomas Noll and Stefan Rieger. Composing Transformations to Optimize Linear Code
Thank you for your attention!

(FM 2008) Thomas Noll and Stefan Rieger. *Verifying Dynamic Pointer-Manipulating Threads*

(ICGT 2008) Stefan Rieger and Thomas Noll. *Abstracting Complex Data Structures by Hyperedge Replacement*


(ICTAC 2007) Thomas Noll and Stefan Rieger. *Composing Transformations to Optimize Linear Code*
Development of HRGs

Program / Data Structure

Construct HRG generating DS

Transform HRG to heap abstraction grammar

Test HRG with program / suitability check

failure

success

HRG suitable for DS / usable to verify program

Check for problems + develop abstraction idea

Add new rules to HRG (+ abstraction-only rules)

Optimization phase Add abstraction-only rules
Partial Concretization II

Solving the Problem

- Enforcing HRGs to be in **apex form** (for all $X \rightarrow H$, the nodes $ext_H$ are only adjacent to terminals)
Partial Concretization II

Solving the Problem

- Enforcing HRGs to be in apex form (for all $X \rightarrow H$, the nodes $\text{ext}_H$ are only adjacent to terminals)

$\Rightarrow$ impractical [Engelfriet, 1992]
Partial Concretization II

Solving the Problem

- Enforcing HRGs to be in **apex form** (for all $X \rightarrow H$, the nodes $\text{ext}_H$ are only adjacent to terminals)

  $\Rightarrow$ impractical [Engelfriet, 1992]

  $\Rightarrow$ Introducing **additional redundant** grammar-rules that do not modify the language
Partial Concretization II

Solving the Problem

- Enforcing HRGs to be in *apex form* (for all \( X \rightarrow H \), the nodes \( ext_H \) are only adjacent to terminals)

\[ \Rightarrow \text{impractical} \ [\text{Engelfriet, 1992}] \]

\[ \Rightarrow \text{Introducing additional redundant grammar-rules that do not modify the language} \]
All $P$-Rules

Concretization from first external vertex

Concretization from second external vertex

Hyperedge Replacement
Abstraction and Concretization
Pointer Logic
Verification and Model Checking
Again the Example
Admissible Results


Approximating the behaviour of graph transformation systems.

Lazy shape analysis.

Extending C for checking shape safety.
In *Graph Transformation for Verification and Concurrency ’05*, volume 154(2) of *ENTCS*, pages 95–112. Elsevier.

A Greibach Normal Form for Context-Free Graph Grammars.


Canonical graph shapes.

Abstract graph transformation.

Separation logic: A logic for shared mutable data structures.