Lecture 3

Towards a Verifying Compiler: Verifying Object Invariants

Wolfram Schulte
Microsoft Research
Formal Methods 2006

Program Invariants, Callbacks,
Aggregates, Ownership, Visibility

Joint work with Rustan Leino, Mike Barnett, Manuel Fähndrich, Herman Venter, Rob DeLine, Wolfram Schulte (all MSR), and Peter Müller (ETH), Bart Jacobs (KU Leuven) and Bor-Yuh Evan Chung (Berkley)
Review: Verification of OO Programs

• What is needed for designing a verifier?

• Which programs can we verify?

• What are the limitations?
Pre- and Postconditions are not Enough

Contracts can break abstraction

```java
class C{
    private int a, z;
    public void M()
        requires a!=0;
        {z = 100/a;}
}
```

We need invariants

```java
class C{
    private int a, z;
    invariant a!=0;
    public void M()
        {z = 100/a;}
}
```
Dealing with Invariants

- *Basic Methodology*
- Object-based Ownership
- Object-oriented Ownership
- Visibility based Ownership
Problem: Reentrancy

class Meeting {
    int day;  int time;
    invariant 0 ≤ day<7 ∧
               day==6 ⇒ 1200<= time;

    void Reschedule(int d )
        requires 0 ≤ d < 7;
    {
        day = d;
        X.P(this);
        if ( day==6 ) time = 1200;
    }
}
Program Model for Object Invariants

- Objects can be valid or mutable
  - \( \text{inv} \in \{ \text{valid}, \text{mutable} \} \) is a new ghost field in each object

- Mutable objects need not satisfy their invariants

- o.inv indicates whether the invariant of o, \( \text{Inv}(o) \), is allowed to be broken

\[
\forall o: \text{o.inv} \neq \text{mutable} \Rightarrow \text{Inv}(o)
\]

Remark: Quantifier ranges over allocated, non-null objects
Field Updates

• Only fields of mutable objects can be updated

\[ \text{Tr}[[o.f = e]] = \]
\[ \text{assert } o \neq \text{null } \land o.inv=\text{mutable}; o.f := e \]
# Pack and Unpack

inv is changed by special source commands

- *unpack*(o) to make o mutable
- *pack*(o) to re-establish invariant and make o valid

<table>
<thead>
<tr>
<th>Tr[unpack o]] =</th>
<th>Tr[pack o]] =</th>
</tr>
</thead>
<tbody>
<tr>
<td>assert o.inv = valid; o.inv := mutable</td>
<td>assert o.inv = mutable; assert Inv(o); o.inv := valid</td>
</tr>
</tbody>
</table>
void Reschedule( int d )
    requires inv==valid ∧ 0≤ d<7;
{
    expose(this){
        day = d;
        if ( day==6 )  time = 1200;
    }
}

Spec# uses expose, defined by
expose(o) s; = unpack o; s; pack o;
Program Invariant

• Theorem (Soundness)

\[ \forall o: \ o.inv \neq \text{mutable} \implies \text{Inv}(o) \]

• Admissible invariants contain only field accesses of the form \( \text{this.f} \)

• Proof sketch
  - **new:**
    new object is initially mutable
  - \( o.f := E; \)
    can only affect invariant of \( o \), asserts \( o.inv = \text{mutable} \)
  - **unpack**(o):
    changes \( o.inv \) to \( \text{mutable} \)
  - **pack**(o):
    asserts \( \text{Inv}(o) \)
Dealing with Invariants

• Basic Methodology
• *Object-based Ownership*
• Object-oriented Ownership
• Visibility based Ownership
Problem: Object Structures

Can we have relax the admissibility condition?

How can we find out that reschedule might break Person’s invariant?

class Person {
    int freeDay;
    Meeting next;
    invariant this.next != null ⇒
        this.next.day != freeDay;
}

class Meeting {
    int day;
    invariant 0 ≤ day < 7;

    void Reschedule(int d ) {
        requires inv==valid;
        {
            expose(this){
                day = d;
            }
        }
    }
}
Invariants in the Presence of Aliasing?

next.day != freeDay

: Person

: Meeting

: Foo

\text{next} \hspace{10pt} \text{next.day} \geq 5

\text{owner} \hspace{10pt} \text{bar} \hspace{10pt} \text{bar.day} \geq 5

\text{call} \hspace{5pt} \text{reschedule}(4)

\text{call} \hspace{5pt} \text{reschedule}(4)

\text{call} \hspace{5pt} \text{reschedule}(4)

\text{inv} = \{ \text{Mutable}, \text{Valid}, \text{Committed} \}
Ownership-Based Invariants

- Establish hierarchy *(ownership)* on objects

- **Ownership rule:** When an object is mutable, so are its (transitive) owners

- An object o may only depend on
  - the fields of o and
  - the fields of objects (transitively) owned by o
Dynamic Ownership

• Each object has a special ghost field, \textit{owner}, that points to its owner object

• \textit{rep}resentation declarations lead to \textit{implicit} \textit{owner invariants}

• \textit{inv} \in \{\text{committed, valid, mutable}\}

• An object is committed, if
  – its invariant is known to hold
  – the owner is not mutable

\begin{verbatim}
class Person {
  int freeDay;
  rep Meeting next;

  /*implicit*/ invariant
  next \neq \text{null} \Rightarrow
  next.owner = this;
...
}
\end{verbatim}
Pack and Unpack with Ownership

- `unpack(o)` and `pack(o)` and change `inv` for `o` and `o`'s rep objects

\[
\text{Tr}[[\text{unpack } o]] = \\
\text{assert } o\text{.inv }= \text{valid}; \\
o\text{.inv }:= \text{mutable}; \\
\text{foreach } (c | c\text{.owner }= o) \\
\quad \{ c\text{.inv }:= \text{valid}; \}
\]

\[
\text{Tr}[[\text{pack } o]] = \\
\text{assert } o\text{.inv }= \text{mutable}; \\
\text{assert } \forall c : c\text{.owner }= o \Rightarrow \\
c\text{.inv }= \text{valid}; \\
\text{foreach } (c | c\text{.owner }= o) \\
\quad \{ c\text{.inv }:= \text{committed}; \}
\]

\[
\text{assert } \text{Inv}( o ); \\
o\text{.inv }:= \text{valid}
\]
Program Invariant with Ownership

Theorem (Soundness)

\[ \forall o: o.inv \neq \text{mutable} \Rightarrow \\
\quad \text{Inv}(o) \land \\
\quad (\forall c: c.owner = o \Rightarrow c.inv = \text{committed}) \]

Admissible invariants contain only field accesses of the form
\[ \text{this.f}_1. \ldots. \text{f}_n \] where \( \text{f}_1. \ldots. \text{f}_{n-1} \) must be rep fields
Method Framing Revisited

*Allow methods to modify also committed objects*

Example. Given

```java
class A{ rep B b;}
class B{ rep C c;}
```

the method

```java
static void m(A a) requires a.inv == valid; modifies a.*;
```

is allowed to modify …

the fields of `a.b` and `a.b.c`

This addresses the transitivity problem of modifies clauses
Method Framing Revisited

*Allow methods to modify also committed objects*

The Post condition for a Spec# modifies W clause

\[
\text{Tr }[[W]] =
(\forall o : \text{ref}, f : \text{field} :: \text{old(Heap}[o,\text{allocated}])
\Rightarrow (o,f) \in \text{old}(W) \lor
\text{old(Heap}[o,f]) = \text{Heap}[o,f] \lor
\text{old(Heap}[o,\text{inv}]) = \text{committed}
\]
class Person {
    int freeDay;
    rep Meeting next;

    invariant next ≠ null ⇒
               next.day ≠ freeDay;

    int doTravel(int td)
        requires inv==valid; modifies this.*;
    { expose(this) {
        freeDay = td;
        if (next!=null) {
            next.reschedule((td+1)%7);
        }
    }
}

class Meeting {
    int day;

    void reschedule( int d )
        requires inv==valid;
    { expose(this)
        day = d;
    }
}
Dealing with Invariants

• Basic Methodology
• Object-based Ownership
• *Object-oriented Ownership*
• Visibility based Ownership
Inheritance

• **Each subtype defines one frame** with its variables
• Single inheritance results in a sequence of frames

<table>
<thead>
<tr>
<th>Example</th>
<th>Objects of</th>
</tr>
</thead>
</table>
| class Cell { int x;  
invariant x>=0;...}  
class BackupCell: Cell {  
rep History h;  
invariant h.last>=x;  
invariant x>10;}  | type Cell have 2 frames:  
[Cell, object]  
type B have 3 frames:  
[BackupCell, Cell, object] |

• **Subtypes are allowed to strengthen invariants**
**Idea: Reduce inheritance to ownership of frames.**

If B is a direct subtype of A, then B has a rep field A. But we only have one inv field, so:

- \( o.inv \) now represents the most derived frame for o which is valid, i.e.
  - \( o.inv <: T \) means Inv(o) holds for the frame T and all its super frames
  - \( o.inv == \text{typeof}(o) \) means Inv(o) holds for all of o’s frames

- \( o.owner \) is now a pair \((p, T)\);
  - p is the owner,
  - T the frame that contains the rep field that points to o.
Refined Representation

Commiting \( c \) to \( p \) means then
\[
c\.commited \iff \text{let } (p,T) = c\.owner \text{ in } p\.inv <: T
\]
Refined Representation

• rep fields f in class T give rise to implicit invariants

\[
\text{invariant } \text{this.f} \neq \text{null } \Rightarrow \text{let } (p, T') = \text{this.f.owner in } p == \text{this } \land T = T'
\]
Pack and Unpack with Inheritance

Given class $T:S$ and $o$ of type $T$. Then

$$\text{Tr}[[\text{unpack}(o \text{ from } T)]] =$$

- assert $o.inv = T$;
- assert $!o.committed$;
- $o.inv := S$
- foreach $(c \mid c.owner===(o,T))$
  - { c.committed := false }

$$\text{Tr}[[\text{pack } o \text{ as } T]] =$$

- assert $o.inv = S$;
- assert $\text{Inv}_T(o)$;
- assert $\forall c: c.owner===(o,T) \Rightarrow c.inv = \text{typeof}(r)$;
- foreach $(c \mid c.owner===(o,T))$
  - { c.committed := true }
- $o.inv := T$

$\text{pack}(o \text{ as } T)$ claims every object that has $o$ as its owner and its rep field declared in $T$
## Inheritance Precondition Problem

**virtual void** Cell.Set(int x)  
requires ...  
modifies this.*;  
{
    unpack(this from Cell);  
    this.x = x;  
    pack(this to Cell);
}

**override void** BackupCell.Set(int x)  
//requires ...  
{
    unpack(this from BackupCell);  
    this.b = this.x;  
    base.Set(x);  
    pack(this to BackupCell);
}

**void** M(Cell c)  
{
    c.Set(23);
}

**How can we verify the dynamically dispatched**  
c.Set call?
Dynamic Dispatch and Preconditions

For virtual methods \( m \), we allow to write the pre:

\[
\text{this.inv} == 1
\]

For each frame in which \( m \) is defined we generate 2 procs

- \( m.C \) is used for \textit{statically bound calls}; its pre:
  \[
  \text{Heap}[o,\text{inv}] = C
  \]

- \( m.C.Virtual \) is used for \textit{dynamically dispatched calls}, its pre:
  \[
  \text{Heap}[o,\text{inv}] = \text{typeof}(o)
  \]

Only \( m.C \) contains the translated code
Inheritance Precondition Example

```
virtual void Cell.Set(int x)
    requires this.inv == 1;
    modifies this.*;
{
    unpack(this from Cell);
    this.x = x;
    pack(this to Cell);
}

void M(Cell c)
    requires c.inv == typeof(Cell);
{
    c.Set(23);
}
```
Dealing with Invariants

• Basic Methodology
• Object-based Ownership
• Object-oriented Ownership
• *Visibility based Ownership*
Rep & Peer Objects

- **Rep fields** are used to **build hierarchical abstractions**.
- **Peer fields** are used to **group sibling objects**.
Rep and Peer Objects

\texttt{class List \{ \\
  rep Node! head; \\
  invariant \\
   (\forall \text{Node } n \mid n.\text{owner} = \text{this} \Rightarrow \\
     \text{next} != \text{null} \Rightarrow \text{next}.\text{prev} = \text{this} \\
     ^\ldots \\
   ) \\
\}}

\texttt{class Node\{ \\
  peer Node next, prev; \\
  invariant \\
   \text{next} != \text{null} \Rightarrow \text{next}.\text{prev} = \text{this} \\
   ^\ldots \\
\}}

Peer fields give rise to additional invariants, for Nodes e.g.

\text{next} != \text{null} \Rightarrow \text{owner} = \text{next}.\text{owner} \\
^\ldots \\
\text{prev} != \text{null} \Rightarrow \text{owner} = \text{prev}.\text{owner}

But how can we reason locally?
Mutually Recursive Object Structures

Objects are mutually dependent
• Objects cannot own each other

```java
class Person {
    Person spouse;

    invariant this.spouse != null ⇒
        this.spouse.spouse = this;

    ...
}
```
Admissible Visibility-Based Invariant

class Person {
    Person spouse dependent Person;

    invariant this.spouse ≠ null ⇒ this.spouse.spouse = this;
    ...
}

The invariant declared in class C may contain a field access R.f iff
- R is “this” or
- R is “this.g₁. ... .gₙ.” and g₁..gₙ are rep or peer fields, and C is mentioned in the dependent clause of f
Proof Obligation for Field Updates

• For a field update o.f := E; we have to prove
  – That o is non-null and mutable (as before)
  – That all other objects whose invariant depends on o.f are mutable

\[
\text{Tr[[o.spouse := E; ]]}
\]

\[
\text{assert } o \neq \text{null } \land o.inv = \text{mutable};
\]

\[
\text{assert } \forall \text{Person } t: t.spouse = o \Rightarrow t.inv = \text{mutable};
\]

• The other objects are determined by \textit{inspecting the invariants} of the all “friend” classes mentioned in the dependent clause (see next slide)
Marriage is Never Easy…

class Person {
    Person spouse dependent Person;
    invariant this.spouse ≠ null ⇒ this.spouse.spouse = this;

    void marry( Person p )
        requires p≠null ∧ p≠this ∧ this.inv = valid ∧ p.inv = valid ∧
        this.spouse = null ∧ p.spouse = null ;
    {
        expose(this)
        expose(p) {
            this.spouse := p;
            p.spouse := this;
        }
    }
}

this.spouse.spouse=this ∧ p.spouse.spouse=p
Summary: Object Invariants

• The methodology solves the problems of
  – Re-entrance (through the explicit inv field)
  – Object structures (through ownership or visibility)

• It can handle
  – Complex object structures including (mutual) recursion
  – Ownership transfer (not shown in this talk)

• The methodology is modular and sound

• Aliasing is not restricted