Static verification of Eiffel programs using Boogie

Julian Tschannen
Chair of Software Engineering
ETH Zürich
Topics

• Introduction to Eiffel and Boogie
• AutoProof
• Translation
  – Types and inheritance
  – Heap model and object creation
  – Routines and frame conditions
  – Generics
  – Polymorphic calls
Introduction to Eiffel

• Object-oriented
• Multiple inheritance
• Generics
• Design by contract
  – Preconditions
  – Postconditions
  – Class invariants
  – Loop invariants
class ACCOUNT
create make
feature
  balance: INTEGER
    make
    do
      balance := 0
    ensure
      balance_set: balance = 0
    end
    deposit (amount: INTEGER)
      require
        amount_not_negative: amount >= 0
      do
        balance := balance + amount
      ensure
        balance_increased: balance = old balance + amount
      end
end
Introduction to Boogie

• Specification language
  – Types
  – Mathematical functions
  – Axioms

• Non-deterministic imperative language
  – Global variables
  – Procedures with pre- and postconditions
  – Control structures (conditional, loop, goto)

• Supports different back-end verifiers (e.g. Z3 or simplify)
type person;
const eve: person;
function age(p: person) returns (int);
function can_vote(p: person) returns (bool);
axiom (age(eve) == 23);
axiom (forall p: person :: can_vote(p) <=> age(p) >= 18);

var votes: int;
procedure vote(p: person);
    requires can_vote(p);
    ensures votes == old(votes) + 1;
    modifies votes;
implementation vote(p: person) {
    votes := votes + 1;
}
AutoProof

- Static verification of a subset of Eiffel
- Part of $EVE^1$ (Eiffel Verification Environment)
- Available online through $Comcom^2$
- Covers:
  - Assignment, conditionals, loops
  - Routine calls, object creation
  - Integer arithmetic, boolean arithmetic
  - Agents, generics
  - Polymorphic calls

(1) http://se.inf.ethz.ch/research/eve
(2) http://cloudstudio.ethz.ch/comcom
AutoProof workflow

- Auto Proof translates Eiffel AST to Boogie
- Boogie generates verification conditions
- SMT solver tries to discharge the VCs
- Result is traced back to Eiffel
Boogie file layout

• Background theory
  – Definitions and axioms

• Classes to be proven
  – Type definition
  – Routine signatures
  – **Routine implementations** (this is proven)

• Referenced routines
  – Routine signature
Demo: Account
Translating Eiffel to Boogie

• Types and inheritance
• Heap model and object creation
• Routines and frame conditions
• Generics
• Loops
• Dynamic contracts
Encoding types

• Boogie type for Eiffel types

```plaintext
type Type;
```

• Type declaration

```plaintext
const unique ACCOUNT: Type;
```

• Encoding inheritance

```plaintext
axiom ACCOUNT <: ANY;
```

• Encoding multiple inheritance

```plaintext
axiom ARRAYED_LIST <: ARRAY;
axiom ARRAYED_LIST <: LIST;
```

```plaintext
class ACCOUNT inherit ANY end
```

```plaintext
class ARRAYED_LIST inherit ARRAY LIST end
```
References and the heap

• Reference type

```plaintext
type ref; const Void: ref;
```

• Generic field type

```plaintext
type Field _;
```

• The heap type is a mapping from references and fields to generic values

```plaintext
type HeapType = <beta>[ref, Field beta]beta;
```

• The heap is a global variable

```plaintext
var Heap: HeapType
```
Ghost fields, functions, attributes

- Ghost field to store allocation status of objects
  
  ```
  const unique allocated: Field bool;
  ```

- Function to declare type of objects
  
  ```
  function type_of(o: ref): Type;
  ```

- Field declaration for each attribute

- Generic field type instantiated with Eiffel type
  
  ```
  const unique field.ACCOUNT.balance: Field int;
  ```

```
class ACCOUNT feature
  balance: INTEGER
end
```
Using the heap

• Functions and axioms using heap

```plaintext
function IsAllocated(heap: HeapType, o: ref) returns (bool);
axiom (forall heap: HeapType, o: ref ::
  IsAllocated(heap, o) <=> heap[o, allocated]);
```

• Assignment to attribute

```plaintext
implementation create.ACCOUNT.make(Current: ref) {
  Heap[Current, field.ACCOUNT.balance] := 0;
}
```
Creating objects on the heap

- Allocate a **fresh** reference on Heap
- Set type and call creation routine

```plaintext
implementation {
  var temp_1: ref;

  entry:
  havoc temp_1;
  assume (temp_1 != Void);
  assume (!Heap[temp_1, allocated]);
  assume (type_of (temp_1) == ACCOUNT);
  Heap[temp_1, allocated] := true;
  call create.ACCOUNT.make(temp_1);
}
```

```plaintext
local
  a: ACCOUNT
do
  create a.make
end
```

\[\begin{align*}
  a & := 7; \quad b := 5 \\
  \text{assert} \quad a &= 7; & \text{assert} \quad b &= 5; & \text{havoc} \quad a; \\
  \text{assert} \quad a &= 7; \\
  \text{assert} \quad a &= 7; & \text{assert} \quad a &\neq 7;
\end{align*}\]
Routine signatures

• Signature consists of
  – Arguments
  – Contracts
  – Frame condition

```
deposit (amount: INTEGER)
  require
    amount >= 0
  do
    ...
  ensure
    balance = old balance + amount
end

invariant
  balance >= 0
```
procedure proc.ACCOUNT.deposit(
    Current: ref,
    arg.amount: int);
// Precondition and postcondition
requires arg.amount >= 0;
ensures Heap[Current, field.ACCOUNT.balance] ==
    old(Heap[Current, field.ACCOUNT.balance]) +
    arg.amount;
// Invariant
free requires Heap[Current, field.ACCOUNT.balance] >= 0;
ensures Heap[Current, field.ACCOUNT.balance] >= 0;
Frame problem

- What can a routine change?

```plaintext
local
  a1, a2: ACCOUNT
do
  create a1.make
  create a2.make
  a1.deposit (100)
  a2.deposit (200)
  check a1.balance = 100 end
  check a2.balance = 200 end
end
```

// create a1, a2
// balance is 0 for both

call ACCOUNT.deposit(a1, 100);
// call ACCOUNT.deposit(a2, 200);
assert 200 >= 0; // pre
h_old := Heap; // store heap
havoc Heap; // invalidate heap
assume Heap[a2, balance] ==
  h_old[a2, balance] + 200; // post
assume Heap[a2, balance] >= 0; // inv

assert Heap[a1, balance] == 100;
assert Heap[a2, balance] == 200;
```
Frame condition

• Describe effect of a routine on heap
• Important for modular proofs

• Different ways to express frame condition
  – Modifies clauses
  – Separation logic
  – Ownership types
  – ...

Modifies clauses in Eiffel

• Not expressible in standard Eiffel
• Special annotation or language extension

```eiffel
deposit (amount: INTEGER)
  note
    modify: balance
  require
    amount >= 0
  ensure
    balance = old balance + amount
  modify
    balance
end
```

• Automatic extraction of modifies clause
  – All attributes mentioned in postcondition
Encoding frame conditions

• Modify whole heap
• Express unchanged parts for each routine

```
procedure proc.ACCOUNT.deposit(
    Current: ref, arg.amount: int);
modifies Heap;
ensures (  
    forall<alpha> $o: ref, $f: Field alpha ::
      ($o != Void &&
       IsAllocated(old(Heap), $o) &&
       (!$o == Current && $f == field.ACCOUNT.balance))
    ==>  
      (old(Heap)[$o, $f] == Heap[$o, $f])
  );
```
Pure functions

- Functions which have no side-effects
- Partial automation of detecting pure functions
  - Each function that is used in a contract
- Functions can be marked as pure
- Purity is checked by Boogie
- Simple encoding

```plaintext
procedure proc.ARRAY.length(Current: ref)
    modifies Heap;
    ensures Heap == old(Heap);
```
Generics

• Distinguish between definition of generic classes and use of generic routines

• Replace generics with a semantic equivalent
  – For each generic class, replace generic parameter with its constraint
  – For each generic routine, create routine signature for each derivation used
  – When a generic routine is used, use signature of specific derivation
Generic classes

class CELL [G -> ANY]
feature
  item: G
set_item (a_item: G)
  do
    item := a_item
  ensure
  end
end

class CELL
feature
  item: ANY
set_item (a_item: ANY)
  do
    item := a_item
  ensure
    item = a_item
  end
end
Generic routines used

```
local
    l_cell1: CELL [STRING]
    l_cell2: CELL [INTEGER]

do
    create l_cell1; l_cell1.set_item ("abc")
    create l_cell2; l_cell2.set_item (7)
end

procedure proc.CELL#STRING#.set_item(
    Current: ref,
    arg.a_item: int
);
ensures Heap[Current, field.CELL#INTEGER#.item] == arg.a_item;
modifies Heap;
ensures <<frame condition>>;
```
Polymorphic calls

- Dynamic type might have different contract than static type
  - Weaker precondition
  - Stronger postcondition
- If dynamic type is known, we can use the **dynamic contract** for the proof
- We use **uninterpreted functions** to encode dynamic contracts
Motivating example

• Strategy pattern

• Implementations of *execute* strengthen postcondition to express their behavior
Demo: Strategy Pattern
Encoding parent postcondition

• Define uninterpreted function

• Link function to actual postcondition depending on type

```plaintext
function post.STRATEGY.execute(h1, h2, current)
    returns (bool);

procedure proc.STRATEGY.execute(Current: ref);
    ensures post.STRATEGY.execute(
        Heap, old(Heap), Current)

axiom (forall h1, h2, current ::
    type_of(current) <: STRATEGY ==> 
    (post.STRATEGY.execute(h1, h2, current) ==> <<parent postcondition>>));
```
Encoding child postcondition

- Link function for parent postcondition to strengthened postcondition for child type

```
axiom (forall h1, h2, current ::
    type_of(current) <: STRATEGY_A ==>
    (post.STRATEGY.execute(h1, h2, current) ==> <<child postcondition>>));
```

- For a child object, the postcondition will imply both postconditions

```
class STRATEGY_A inherit STRATEGY
    feature
        execute do
            ...
        ensure
            <<child postcondition>>
    end
end
```
Encoding dynamic preconditions

- Inverse implication: actual precondition implies precondition function

```plaintext
function pre.STRATEGY.execute(h1, current) returns (bool);

procedure proc.STRATEGY.execute(Current: ref);
  requires pre.STRATEGY.execute(Heap, Current)

axiom (forall h1, current ::
  type_of(current) <: STRATEGY ==> (
    <<parent precondition>> ==> pre.STRATEGY.execute(h1, current) ));
```
implementation {
  var s: ref;
entry:
  assume Heap[s, $allocated] && s != Void;
  assume type_of(s) == STRATEGY_A;

  // call proc.STRATEGY.execute(s);
  assert pre.STRATEGY.execute(Heap, s);
  h_old := Heap;
  havoc Heap
  assume «<frame condition>>; // relates Heap to h_old
  assume post.STRATEGY.execute(Heap, h_old, s);

  assert «<child postcondition>>;
}

axiom (forall h1, h2, current ::
  type_of(current) <: STRATEGY_A ==> (post.STRATEGY.execute(h1, h2, current) ==> «<child postcondition>>));
Conclusions

• Automatic verification of object-oriented programs achieved through an intermediate verification language

• Different ways of translation
  – Mapping Eiffel semantics to Boogie
  – Eiffel side source-to-source translation

• Modularity of proofs allows to partially prove a program