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)
Boogie: Code example

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 (Eiffel Verification Environment)
• Covers:
  – Assignment, conditionals, loops
  – Routine calls, object creation
  – Integer arithmetic, boolean arithmetic
  – Agents, generics
  – Polymorphic calls
AutoProof workflow

- Translates AST from EiffelStudio to Boogie
- Uses Boogie verifier to check Boogie files
- Traces verification errors back to Eiffel source
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
• Dynamic contracts
Encoding types

• Boogie type for Eiffel types

```boogie
type Type;
```

• Type declaration

```boogie
const unique ACCOUNT: Type;
```

• Encoding inheritance

```boogie
axiom ACCOUNT <: ANY;
```

• Encoding multiple inheritance

```boogie
axiom ARRAYED_LIST <: ARRAY;
axiom ARRAYED_LIST <: LIST;
```
References and the heap

• Reference type

type ref;  const Void: ref;

• Generic field type

type Field _;

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

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

• The heap is a global variable

var Heap: HeapType
Ghost fields and attributes

• Ghost field to store allocation status of objects

```plaintext
const unique $allocated: Field bool;
```

• Ghost field to store type of objects

```plaintext
const unique $type: Field Type;
```

• Field declaration for each attribute

• Generic field type instantiated with Eiffel type

```plaintext
const unique field.ACCOUNT.balance: Field int;
```
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;
}
```

```plaintext
make
    do
        balance := 0
    end
```
Creating objects on the heap

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

```plaintext
implementation {
    var temp_1;

    entry:
    havoc temp_1;
    assume (temp_1 != Void) &&
    (!Heap[temp_1, $allocated]);

    Heap[temp_1, $allocated] := true;
    Heap[temp_1, $type] := ACCOUNT;
    call create.ACCOUNT.make(temp_1);
}

local
    a: ACCOUNT

do
    create a.make
end
```
Routine signatures

• Signature consists of
  – Arguments
  – Contracts
  – Frame condition

\[
\text{deposit (amount: INTEGER)} \\
\hspace{1em} \text{require} \\
\hspace{2em} \text{amount} \geq 0 \\
\hspace{1em} \text{do} \\
\hspace{2em} \ldots \\
\hspace{1em} \text{ensure} \\
\hspace{2em} \text{balance} = \text{old balance} + \text{amount} \\
\hspace{1em} \text{end} \\
\text{invariant} \\
\hspace{1em} \text{balance} \geq 0
\]
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 condition
    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]));
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
  - ...
Frame condition in Eiffel

• Modifies clauses
  – What attributes a routine may modify
  – Needs change to Eiffel language

```
deposit (amount: INTEGER)
  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

```plaintext
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

```plaintext
class CELL [G -> ANY]
feature
  item: G
  set_item (a_item: G)
    do
      item := a_item
      ensure
        item = a_item
    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
Encoding parent postcondition

- Define uninterpreted function for postcondition
- Link function to actual postcondition depending on type

```latex
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 ::
  h1[current, $type] <: STRATEGY ==> 
  (post.STRATEGY.execute(h1, h2, current) ==> <<parent postcondition>>))
```
Encoding child postcondition

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

\[
\text{axiom (forall } h1, h2, \text{ current :: h1[current, $type]} <: \text{STRATEGY_A} \implies \left( \text{post.STRATEGY.execute(h1, h2, current)} \implies \text{<<child postcondition>>} \right));
\]

• For a child object, the postcondition function will imply both postconditions
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 ::
    h1[current, $type] <: STATEGY ==>
    (<<parent precondition>> ==>
        pre.STRATEGY.execute(h1, current) ));
```
Call site example

implementation {
  var s: ref;
entry:
  assume Heap[s, $allocated] && s != Void;
  assume Heap[s, $type] == STRATEGY_A;

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

  assert <<child postcondition>>;
}

axiom (forall h1, h2, current ::
  h1[current, $type] <: STRATEGY_A ==> (post.STRATEGY.execute(h1, h2, current) ==> <<child postcondition>>));
Conclusions

• Automatic verification of object-oriented programs using 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
Theses

• You can do your master’s thesis, bachelor’s thesis or a semester thesis with us

• Topics:
  – Static and dynamic verification of object-oriented programs
  – Specification of programs

• Contact me:

  julian.tschannen@inf.ethz.ch