



Static verification of Eiffel programs using Boogie

Julian Tschannen

Chair of Software Engineering

ETH Zürich



- Overview
- Translation
 - Types and inheritance
 - Heap model and object creation
 - Routines and frame conditions
 - Generics
 - Dynamic contracts



- Static verification of a subset of Eiffel
- Part of EVE (Eiffel Verification Environment)
- Covers:
 - Assignment, loops, routine calls, object creation
 - Integer arithmetic, boolean arithmetic
 - Agents
- Not covered:
 - Exception handling, once routines
 - Expanded types, floating point arithmetic

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
- Proven classes
 - Type definition
 - Routine signatures
 - **Routine implementations (this is proven)**
- Referenced routines
 - Routine signature



-
- User interface
 - Account
 - Generated Boogie file

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

```
type Type;
```

- Type declaration

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

- Encoding inheritance

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

- Encoding multiple inheritance

```
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

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

- Ghost field to store type of objects

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

- Field declaration for each attribute
- Generic field type instantiated with Eiffel type

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

Using the heap



- Functions and axioms using heap

```
function IsAllocated(heap: HeapType, o: ref)
  returns (bool);

axiom (forall heap: HeapType, o: ref ::
  IsAllocated(heap, o) <==> heap[o, $allocated]);
```

- Assignment to attribute

```
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

```
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);
}
```

Routine signatures



- Signature consists of
 - Arguments
 - Contracts
 - Frame condition

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

invariant
  balance >= 0
```

Encoding routine signatures



```
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

```
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

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



- 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
```

```
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  
);
```

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

Dynamic contracts

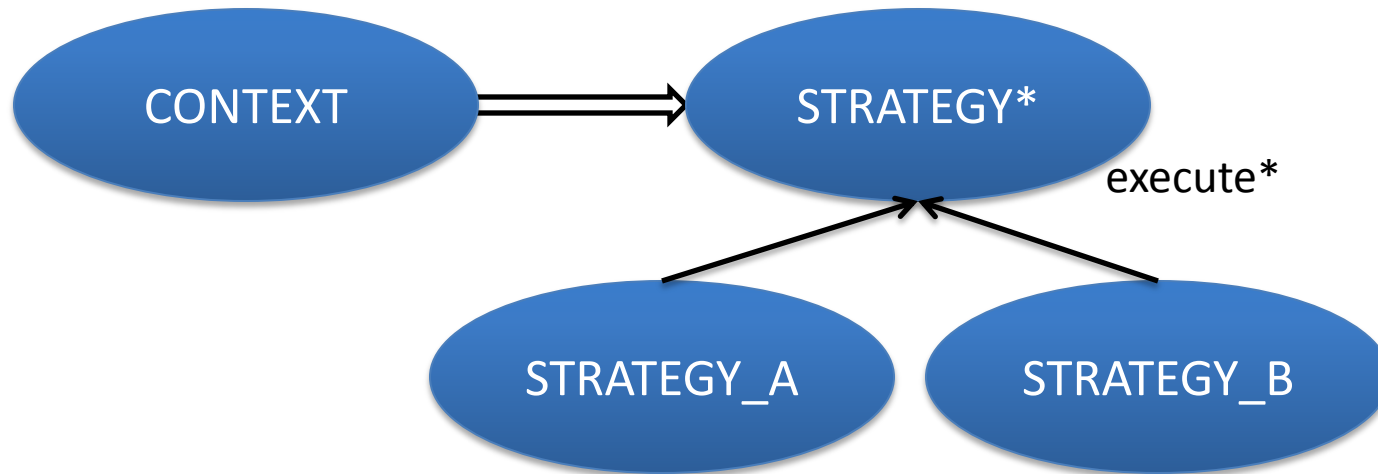


- Dynamic type might have different contract than static type
 - Weaker precondition
 - Stronger postcondition
- If dynamic type is known, we can use the **dynamic contract** for the proof
- We use **abstract predicates** to encode dynamic contracts

Motivating example



- Strategy pattern



- Implementations of *execute* strengthen postcondition to express their behavior
- Demo

Encoding parent postcondition



- Define abstract postcondition predicate
- Link predicate to actual postcondition depending on type

```
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 predicate for parent postcondition to strengthened postcondition for child type

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

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

Encoding dynamic preconditions



- Inverse implication: actual precondition implies precondition predicate

```
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] <: STRATEGY ==>
  (<<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>>;
}
```

Conclusions

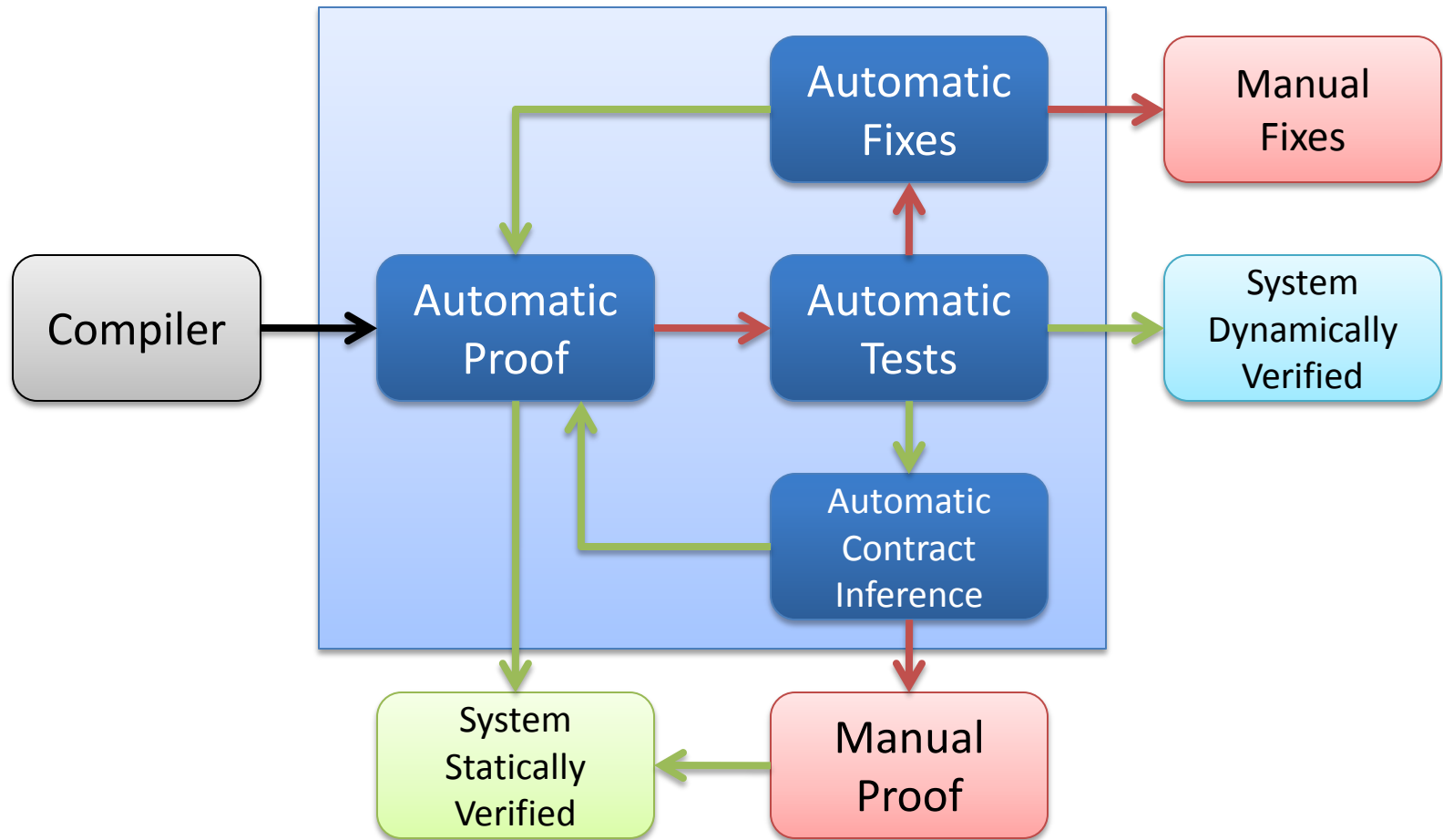


- Different ways of translation
 - Mapping Eiffel semantics to Boogie
 - Eiffel side source-to-source translation
- Needs full translation to prove full programs
 - Expanded types missing
 - Exceptions missing
- Modularity of proofs allows to partially prove a program

Verification assistant



- Continuous verification and code inference





- You can do your master's thesis, bachelor's thesis or a semester thesis with us
- Topics:
 - **Verification of object-oriented programs**
 - **Verification assistant**
- Contact me:
julian.tschannen@inf.ethz.ch