PW: Final Write-up

1. Overview

This document summarizes the transformation of while loop specifications in Viper. Two approaches are compared—the InhaleExhale (IE) transformation and a Recursive method transformation. We discuss the verification speed and overhead, highlight observed bugs, detail the transformation process, and list future work and extension goals.

The while loop with specifications, a basecase and ghost code can now be desugared in 2 different ways which have their own strengths and expressive powers. Both versions can be compiled in separate .jars which can then be used seamlessly in VSCode.

2. Verification Speed and Overhead Comparison

InhaleExhale Transformation

For each while loop with a specification, the IE approach introduces the following constructs:

- **Labels & Copies:** 3 labels and 3 copies of all loop targets (typically 1–5 targets).
- **Havocs:** 2 havocs applied to all targets and 1 havoc method per distinct target type.
- **Inhales/Exhales:** 4 exhales and 3 inhales (applied to the pre- and post-conditions).
- **Loop Structure:** A `while(true)` loop with one non-deterministic Boolean variable and 2 nested if-else branches.
- **Code Blocks:** Separate Ghost code, Basecase code, and Body code.

Recursive Transformation

The recursive approach, which leverages local reasoning via helper method calls, introduces significantly fewer constructs:

- **Helper Methods:** 2 calls to a helper method and one helper method declaration (with targets and variables as parameters and targets as return values).
- **Copies & Branching:** 1 copy of all targets and 1 if-else branch.
- **Code Blocks:** Ghost code, Basecase code, and Body code are all present.

Summary:
The Recursive version produces much less translated code. Its simplicity leads to fewer copies and less overhead, at the cost of some expressiveness. Indeed, although the recursive method has verified successfully many test cases (e.g., `inc.vpr`, `len.vpr`, `list_copy.vpr`) it does fail on the following test case.

Difference in Expressive Power

Original Test Case (“only_works_for_ie.vpr”)

```
method f(){


   var i : Int := 1


   var cond : Bool := true
   while(cond)

       ensures i == 1

     {

       cond := false

   }


}
```

IE version (which verifies successfully)

```
method f()

{

 {

  var i: Int

  var cond: Bool

  i := 1

  cond := true

  {

   var _nondet: Bool

   var pre_loop_cond: Bool

   label pre_loop

   pre_loop_cond := cond

   exhale true

   cond := havoc_Bool()

   if (_nondet) {
        while (true) {
          var pre_iteration_cond: Bool
          inhale true
          label pre_iteration
          pre_iteration_cond := cond
          if (cond) {
            var after_iteration_cond: Bool
            cond := false
            label after_iteration
            after_iteration_cond := cond
            exhale true
            cond := havoc_Bool()
            inhale i == 1
            exhale i == 1
          } else {
            exhale i == 1}
        }
      } else {
        inhale i == 1}
    }
  }
}

method havoc_Bool() returns (x: Bool)
```

Recursive Version (throws an error, see below for more detail)

```
method f()
{
  {
    var i: Int
    var cond: Bool
    i := 1
    cond := true
    cond := HELPER_inductive_step(i, cond)
  }
}


method HELPER_inductive_step(i: Int, cond_init: Bool) returns (cond: Bool)
  requires true
  ensures i == 1
{
  cond := cond_init
  inhale cond
  cond := false
  cond := HELPER_inductive_step(i, cond)
}


method HELPER_basecase(i: Int, cond_init: Bool) returns (cond: Bool)
  requires true
  ensures i == 1
{
  cond := cond_init
  inhale !cond
}
```

Error (highlighted in red):

“Postcondition i == 1 might not be preserved in base case. Assertion i == 1 might not hold” (only_works_for_ie.vpr@8.17--8.23)

The error is because the recursive version doesn’t have any information about `i`. It only knows what the precondition can tell it. Here the precondition is useless so the basecase fails.
The reason that it works for the inductive step is that there we don’t have to prove the postcondition, we only need to prove that the postcondition of the next iteration proves the postcondition of the current iteration.

Please note that the error correctly points back to the original program’s while loop’s postcondition “`i == 1`”.

This test case works in the IE version, because there we have the information `i := 1` in the outer scope of the desugared program, so it has access to this without us having to include it in the precondition.

Conclusion

This reveals one of the major benefits of the IE version, being that each augmented while loop’s knows about the outer scope of where they are defined; whereas the recursive version’s 2 methods only know about their specs (which is not as much of a problem for the inductive step as it is for the basecase).

3. Example Comparison (“not_missing_pre.vpr”)

This is to showcase both versions’ overhead and how differently they work to achieve the same goal of verifying a while loop with specifications.

Original Program (before transformation)

```
method f(l : Ref)
   requires l != null
{
   var curr : Ref := l;
   var i : Int := 1;

   while(curr != null)
       ensures curr == null ==> pre(i) < i
   {
       curr := null;
       ghost{
           i := i + 1;
       }
   }
   basecase{ // curr == null && i = pre(i) + 1
       i := i + 1;
   }
}
```

Inhale Exhale (IE) Version

Viper

```
method f(l: Ref)
  requires l != null
{
  {
    var curr: Ref
    var i: Int
    curr := l
    i := 1
    {
      var _nondet: Bool
      var pre_loop_i: Int
      var pre_loop_curr: Ref
      label pre_loop
      pre_loop_i := i
      pre_loop_curr := curr
      exhale true
      i := havoc_Int()
      curr := havoc_Ref()
      if (_nondet) {
        while (true) {
          var pre_iteration_i: Int
          var pre_iteration_curr: Ref
          inhale true
          label pre_iteration
          pre_iteration_i := i
          pre_iteration_curr := curr
          if (curr != null) {
            var after_iteration_i: Int
            var after_iteration_curr: Ref
            curr := null
            label after_iteration
            after_iteration_i := i
            after_iteration_curr := curr
            exhale true
            i := havoc_Int()
            curr := havoc_Ref()
            inhale curr == null ==>
              old[after_iteration](after_iteration_i) < i
            i := i + 1
            exhale curr == null ==>
old[pre_iteration](pre_iteration_i) < i
          } else {
            i := i + 1
            exhale curr == null ==>
old[pre_iteration](pre_iteration_i) < i
          }
        }
      } else {
        inhale curr == null ==> old[pre_loop](pre_loop_i) < i}
    }
  }
}

method havoc_Ref() returns (x: Ref)


method havoc_Int() returns (x: Int)
```

Recursive Version

viper

```
method f(l: Ref)
  requires l != null
{
   var curr: Ref := l;
   var i: Int := 1;
   i, curr := HELPER_inductive_step(i, curr);
}

method HELPER_inductive_step(i_init: Int, curr_init: Ref)
  returns (i: Int, curr: Ref)
  requires true
  ensures true && (curr == null ==> old(i_init) < i)
{
  i := i_init;
  curr := curr_init;
  inhale curr != null;
  curr := null;
  i, curr := HELPER_inductive_step(i, curr);
  i := i + 1;
}

method HELPER_basecase(i_init: Int, curr_init: Ref)
  returns (i: Int, curr: Ref)
  requires true
  ensures true && (curr == null ==> old(i_init) < i)
{
  i := i_init;
  curr := curr_init;
  inhale !(curr != null);
  i := i + 1;
}
```

In the recursive approach, much of the overhead is hidden inside helper calls, reducing the overall code size and complexity.
The havoc methods, which are necessary for the IE version, are completely useless in the Recursive version which havocs implicitly each method's parameters (the method `HELPER_basecase` has no information about `i_init` for example apart from whatever the precondition might give it, so it effectively havocs `i_init`)

4a. Bug Report

Bug 1: Permission Reporting Misplacement

viper

```
// SPECS, G, BC, PRE

field next : Ref

predicate List(l : Ref) {
  (l != null) ==> acc(l.next) && List(l.next)
}

method add_iterative_spec(l : Ref)
  requires List(l)
{
  var curr : Ref
  curr := l

  while(curr != null)
     requires List(curr)
       //:: ExpectedOutput(not.wellformed:insufficient.permission)
```

```
    //::
ExpectedOutput(precondition.not.established:insufficient.permission)
    requires List(curr.next)
  {
  }
}
```

- **Intended Behavior**
  Two expected outputs (for demonstration/testing):
  1. `//:: ExpectedOutput(not.wellformed:insufficient.permission)`
  2. `//:: ExpectedOutput(precondition.not.established:insufficient.permission)`

Actual Behavior
The tool produces:
viper

```
while(curr != null)
//:: Output(not.wellformed:insufficient.permission)
    requires List(curr)

    //:: Output =
ExpectedOutput(precondition.not.established:insufficient.permission)
    requires List(curr.next)
{
}
```

- The error message of the first error (insufficient permission) points to `List(curr.next)` logically, but the reported position highlights `List(curr)`.

Attempted Fix
Tried adding (LoopSpecsRec.scala 141:142 and 155:156):
viper

```
//          case ContractNotWellformed(offendingNode, reason, cached)
=>
//            ContractNotWellformed(offendingNode.withMeta(reason.pos,
NoInfo, reason.offendingNode.errT), reason, cached)
```

to transform the error (similar to the successful handling of:
viper
```
case PostconditionViolated(offNode, member, reason, cached) =>
  PostconditionNotPreservedInductiveStep(offNode, reason, cached)
```

- LoopSpecsRec.scala 138:139), but it does not resolve the issue.

Desugared Code Example
plaintext
CopyEdit
```
field next: Ref

predicate List(l: Ref) {
  l != null ==> acc(l.next, write) && acc(List(l.next), write)
}

method add_iterative_spec(l: Ref)
  requires acc(List(l), write)
{
  {
    var curr: Ref
    curr := l
    HELPER_inductive_step(curr)
  }
}

method HELPER_inductive_step(curr: Ref)
  requires acc(List(curr), write) && acc(List(curr.next), write)
  ensures true
{
  inhale curr != null
  HELPER_inductive_step(curr)
}

method HELPER_basecase(curr: Ref)
  requires acc(List(curr), write) && acc(List(curr.next), write)
  ensures true
{
  inhale !(curr != null)
}
```

Silicon Output:

less
CopyEdit

```
Silicon found 2 errors in 6.68s:
  [0] Contract might not be well-formed. There might be insufficient
permission to access curr.next
(recreating_precond_bug_minimal.vpr@17.16--17.26)
  [1] Precondition curr.next might not hold on entry. There might be
insufficient permission to access curr.next
(recreating_precond_bug_minimal.vpr@21.21--21.30)
```

Suspected Cause

Using `e1.pos` in the code snippet below ("LoopSpecsPASTExtension.scala" 202:212)may lead to misalignment in reported positions:

scala

```scala
private def conjoin_exps(exp : PDelimited[PSpecification[_],
Option[PSym.Semi]], t : Translator) = {
 val exps = exp.toSeq.map(spec => t.exp(spec.e))

 exps match {
   case Seq() => TrueLit()() // Empty list
   case Seq(single) => single // Exactly one expression
   case head +: tail => // More than one expression
     tail.foldLeft[Exp](head) { (e1, e2) => And(e1, e2)(e1.pos) }
 }

}
```

Bug 2: VSCode vs. IntelliJ/SiliconRunner Discrepancies

There seems to be a mismatch between errors in my Intellij/ SiliconRunner env and VSCode. For the file *"inc_wrong_precond.vpr"* where I add **"requires List(curr.next)"** the annotation **"//:: ExpectedOutput(precondition.not.established:insufficient.permission)"** is placed just above this line and this output is indeed what happens. I also confirmed this by simply running the file with SiliconRunner.

The error in both cases is:
"[0] Precondition curr.next might not hold on entry. There might be insufficient permission to access curr.next (inc_wrong_precond.vpr@32.21--32.30)"

However, in VSCode, the highlighting goes over **List(curr)** for some reason. And the error is:
"Exhale might fail. There might be insufficient permission to access curr.next (inc_wrong_precond.vpr@30.16--30.26)"

It seems that in VSCode the error transformation defined with

scala

```
case ExhaleFailed(offNode, reason, cached) =>

  PreconditionNotEstablished(offNode, reason, cached)
```

never happened.

Excerpt of the Offending File

viper

```
“// SPECS, G, BC, PRE



field val : Int

field next : Ref



predicate List(l : Ref) {

  (l != null) ==> acc(l.val) && acc(l.next) && List(l.next)

}
```

```
function values(l: Ref): Seq[Int]

   requires List(l)

{

   unfolding List(l) in

   l == null

       ? Seq()

       : Seq(l.val) ++ values(l.next)

}



method add_iterative_spec(l : Ref)

  requires List(l)

  ensures List(l)

  ensures |values(l)| == |old(values(l))|

  ensures forall i : Int :: (0 <= i < |values(l)|) ==> values(l)[i]
   == old(values(l)[i]) + 1

  {

  var curr : Ref

  curr := l
```

```
while(curr != null)

   requires List(curr)

     //:: ExpectedOutput(not.wellformed:insufficient.permission)




   //::
 ExpectedOutput(precondition.not.established:insufficient.permissi
 on)

   requires List(curr.next)

   ensures List(pre(curr))

    && |values(pre(curr))| == |pre(values(pre(curr)))|

     && forall i : Int ::

          (0 <= i < |values(pre(curr))|) ==> values(pre(curr))[i]
 == pre(values(pre(curr))[i]) + 1

{

   unfold List(curr)

   curr.val := curr.val + 1

   curr := curr.next

   ghost{

      fold List(pre(curr))

   }

}
```

```
      basecase{

         fold List(pre(curr))

      }






    }

    “
```

In the IE version, the above test case is the only one where something differs between VSCode and Intellij.

Further Observations

But more generally, for the Recursive Version, it seems that the error reporting is off in VSCode, the error usually has *“NoPosition”*. This doesn’t happen in the IE version. The positive examples verify without any issues; whereas some of the negative examples treat the errors in a strange way different from how it happens while running *“SiliconRunner.scala”* in IntellijIDEA for example.

This happens with the following test files in the “loopspecsrec/” folder:

- *“prec_not_established.vpr”*
- *“prec_not_preserved.vpr”*
- *“inc_wrong_precond.vpr”*: The contract error gets incorrectly mapped to “List(curr)” as with IE but additionally, the precondition might not hold error has NoPosition.
- *“recreating_precond_bug_minimal.vpr”* which is the minimal version of the above test file “inc_wrong_precond.vpr”

Summarizing Observations

1. The **positive examples** generally verify without issues across both IDEs.
2. The **negative examples** (only the 4 above) expose inconsistencies in error reporting.
3. **Possible Explanation:**
   a. **Marco "Hm, interesting. VSCode runs Silicon through ViperServer. It should do the same error transformations, but the code is different, so it's possible there's some issue there."**

4b. Weird Behavior

1. Pre

With `parent` being a Ref, and `l` being a field.

```
acc(pre(parent).l)
```

⇒

```
acc(old[pre_loop](pre_loop_parent).l, write)
```

Whereas:

```
acc(pre(parent.l))
```

Isn't licit because `parent.l` is a field and `pre()` expects a Ref.

And

```
pre(acc(parent.l))
```

Isn't licit because `acc(.)` is impure and `pre()` expects a pure expression.

2. Ghost Scope vs Body Scope

The ghost scope is independent of the loop body's scope. It cannot access any variables declared in the body. It can of course access the loop's outer scope.

5. Transformation Overview: Viper to Viper

The transformation process targets the desugaring of while loop specifications.
Two key transformations are used.
Here we formalize them.

Assumptions for both transformations:

1. No `goto` statements.
2. No explicit `old` constructs beyond those introduced by the transformation.

Pre Desugaring (in the postcondition, the ghost code and the base case code):

This is how pre(.) is desugared:

a. **`pre(var)`:**
   Represents the variable state at the start of an iteration. It is desugared into a copy of that variable.
b. **`pre(heap_state)`:**
   Represents the heap state at the start of an iteration. It is desugared into `old[heap_state]`.

InhaleExhale Transformation:

Original Structure:

```
c_before


While(cond)

Requires Pre(vars)

Ensures Post(vars, pre(vars))

{
    c_body

    ghost{
       c_ghost
    }
}

basecase{
   c_basecase
}

c_after
```

Transformed Structure:

```
c_before


Label pre_loop
vars_pre_loop  (Copies of all targets: body + basecase)

Exhale Pre(vars)
Havoc targets

If(*) {
    while(true) {
        Inhale Pre(vars)
        Label pre_iteration
        vars_pre_iteration  (Copy of targets)

        if (cond) { // Inductive Step
            c_body
            Label after_iteration
            vars_after_iteration (Copy of targets)
            Exhale Pre(vars)
            Havoc targets
            Inhale Post(pre(expr) -> old[after_iteration](expr))
            c_ghost
            Exhale Post(pre(expr) -> old[pre_iteration](expr))
        } else { // Base Case
            c_basecase
            Exhale Post(pre(expr) -> old[pre_iteration](expr))
        }
    }
} else {
    Inhale Post(pre(expr) -> old[pre_loop](expr))
}

c_after
```

Recursive Method Transformation

Original Structure:

```
Main {
    c_before
    While(cond)
       Requires Pre(vars)
       Ensures Post(vars, pre(vars))
    {
       c_body
       ghost{
           c_ghost
       }
    }
    basecase{
       c_basecase
    }
    c_after
}
```

Transformed Structure:

```
main {
    c_before
    targets := HELPER_inductive_step(targets, vars)
    c_after
}

// For a target called ‘x’:
//
//                            x_init                        x
method HELPER_inductive_step(targets_init, vars) returns (targets):

Requires Pre(vars, targets_init)

Ensures Post(vars, targets, pre(targets))
// targets here is after the entire method

Where `pre(.)` is desugared according to case:

- `pre(target)` → `target_init`;
- `target` → `target` (target at end of loop iteration)

{

    targets := targets_init


    inhale cond

    c_body
    (body modifies targets, as it can’t modify method arguments)
     targets :=helper(targets, vars)
     //vars haven't changed

     c_ghost
     (can modify targets)

}


// For a target called ‘x’:
//
//                         x_init                      x
method HELPER_basecase(targets_init, vars) returns (targets):

Requires Pre(vars, targets_init)

Ensures Post(vars, targets, pre(targets))
// targets here is after the entire method

Where `pre(.)` is desugared according to case:

- `pre(target)` → `target_init`;
- `target` → `target` (target at end of loop iteration)

{

    targets := targets_init


    inhale !cond
    c_basecase
    (can modify targets)

}
```

---

6. PW Meeting Notes

- **Keyword Renaming:**
  Proposed alternatives for block names:
  - **For Ghost Block:** Options include `unwind`, `resume`, `revert`, `upcall`, or `rebind`.
  - **For Base Case Block:** Options include `base`, `done`, `exit`, `final`, or `conclude`.
- **Inhale and Magic Wand:**
  Discussion on the possibility of rewriting specifications into a magic wand formulation. One idea is to express the postcondition as:
  `(old state) current postcondition --*> (initial state) previous postcondition.`
  But generalizing postcondition expressions to magic wand variables is an open problem.
- **Extension Goals:**
  - Lifting while loop specifications to high-level languages (e.g., Java) rather than Viper.
  - Labeling code sections to allow alternative transformations.
  - Investigating a magic wand-based transformation for loop specifications.

---

7. Future Extension Work (taken from the PW initial write-up)

• Add native support inside Viper's verification backends (one or both) for pre-post constructs. We expect this to provide benefits in contrast to the above rewriting scheme (see below for additional information on these benefits).

• Evaluate the implementations according to the two different Viper verification backends: Carbon vs Silicon

• Compare and Contrast with a magic wand implementation

• Prove formally that a loop with pre-post annotations functionally behaves as a recursive method. Examples of possible mismatches: permission introspection...

8. Running Instructions

I usually run “SiliconRunner.scala” in IntellijIDEA with the following “Program Arguments”:
“--plugin=viper.silver.plugin.standard.loopspecs.LoopSpecsPlugin
--printTranslatedProgram
--numberOfErrorsToReport=0
silver/src/test/resources/loopspecsie/test_file.vpr”

You can replace the plugin with:
“--plugin=viper.silver.plugin.standard.loopspecs.LoopSpecsPluginRec” to run the recursive version instead of the IE version.

Furthermore, “--printTranslatedProgram” is a useful way to see the transformed program and can bring valuable information for bug-fixing or verifying a complex input program.
Finally, “--numberOfErrorsToReport=0” forces Silicon to report all errors which is useful for catching all errors in a test file with multiple “//:: ExpectedOutput(error:reason)”.