Chair of
Software Engineering

Loop invariant inference from postconditions in EVE

Bachelor Thesis

By:	Michael Ameri
Supervised by:	Carlo Alberto Furia
	Julian Tschannen
	Prof. Dr. Bertrand Meyer

Student Number: 08-918-435

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

inf Informatik
Computer Science

Contents

Abstract

Loop invariants play a major role in automatic verification of programs. Finding a suiting loop invariant can however be rather difficult. This thesis presents an implementation of automatic loop invariant inference based on postcondition mutation integrated directly within EVE. EVE is a development environment built on top of EiffelStudio. It integrates verification techniques within the IDE. Autoproof is a tool within EVE which delivers static verification based on Hoare-style proofs using Eiffel's contracts. The implementation presented in this thesis provides an additional tool within AutoProof, so that a developer can execute the postcondition mutation algorithm and obtain invariants without having any further knowledge about the implementation.

1 Introduction

To prove that a feature is correct, it must be shown that the feature terminates, that for each feature which is called by this feature, its pre-conditions are not violated, and that once the called feature terminates, its postconditions are fulfilled. No universal algorithm can exist which would perform this proof for every feature. Having invariants attached to loops can help automate this proof in various cases. However, finding a loop invariant can be rather cumbersome for a developer, which is why they are often omitted in practice. With this thesis I aim to integrate automatic loop invariant generation within the verification environment EVE.

EVE [1] is a development environment built on top of EiffelStudio. It integrates verification techniques within the IDE. Autoproof is a tool within EVE which delivers static verification based on Hoare-style proofs using Eiffel's contracts.[1]
Among other contracts, Eiffel supports the notions of both postconditions for features and invariants for loops. The goal of this thesis is to implement loop invariant inference from postcondition mutation for Eiffel and integrate the solution within the Autoproof mechanism of EVE. To achieve this, the algorithm presented in [2] was adopted. To date, a command line solution which operates on annotated Boogie code is available at [3].
Once a developer using EVE decides to launch the postcondition mutation algorithm presented here, the following is supposed to happen: Postconditions of features that contain loops are mutated based on heuristics to generate invariant candidates for each loop in that feature. The Boogie verifier [4] is then used to test each candidate if it is an invariant of some loop. The candidates which pass the test are then presented in Autoproof's window. This allows developers to add automatically generated invariants to their loops. This procedure offers several advantages to the developer. The loop invariants can help to understand what exactly is being executed. In some instances the automatically generated loop invariants also enable AutoProof to successfully prove that a postcondition of the feature never fails.

This thesis will first explain the motivation behind using the implemented inference techniques in section 2. In the next section the algorithm behind the invariant generation based on postcondition mutation and its implementation are presented. After that section 4 demonstrates how a developer using EVE can execute the automatic invariant generation within the IDE, before section 5 evaluates the implementation based on several test cases. Thereafter a set of API is presented in section 6, which can be used to extend or alter the implemented functionality. In the end a conclusion is drawn and ways to further develop the implementation are presented, both in section 7.

[1] http://se.inf.ethz.ch/research/eve/

2 Motivation behind inferring loop invariants and overall goals

This section discusses the difficulties within automatic loop invariant generation and the reasoning behind using mutated postconditions as loop invariants. Furthermore the heuristics behind the algorithm which is presented in section 3 are explained.

Automatically generating loop invariants is not a trivial task. This implementation focuses on inferring the invariants from the postcondtions of features which contain the loop, as suggested in [2].

By definition, a loop invariant must hold at the following points during execution:

- After the loop is initialized, and before the first execution of the loop;
- After each consecutive execution of the loop's body.

This makes it clear that the loop's invariant cannot be stricter than its postcondition. Therefore trying to infer loop invariants by weakening the loops postcondition is a valid approach. Since Eiffel doesn't have a notion of postconditions for loops, the feature's postconditions are used. This often already yields good results. If the current feature's post conditions don't match those of the loop, a developer who wants to make use of this automatic loop invariant inference can always create a new feature containing just the loop and attach the loops postconditions to that new feature.

An invariant of a loop must not only be weaker than its postcondition, it must be weak enough to hold after the initialization and after each consecutive iteration. It must also be strong enough to yield the postcondition after the last execution of the loop's body. This is why this implementation uses the three heuristics which are presented in section 2.1, to mutate postconditions in anticipation of finding a valid loop invariant.

2.1 Heuristics used for postcondition mutation

2.1.1 Constant relaxation

Constant relaxation is the notion of replacing one or more constants by a variable. In many cases, this leads to a weaker form of the postcondition, which turns out to be a loop invariant. An example of this might be the feature *max_value* which iterates through an array to find the largest element in that array, in each iteration updating the maximum value if necessary. (The feature can be found in the appendix.) The postcondition reads as:

```
across 1 |..| a.count as k all a[k.item] <= max_value end
```

By replacing *a.count*, which is a constant, by the iteration number (denoted *i*), which is a variable, we get a valid loop invariant:

```
across 1 |..| i as k all a[k.item] <= max_value end
```

Since we only look at elements of the array up to the *i*'th position, this is indeed a weakened form of the postcondition, which looks at all elements.

In the algorithm, this heuristic is used in the two features which perform coupled- and uncoupled-mutations.

2.1.2 Uncoupling

This notion is used in the feature which performs uncoupled mutations. Instead of replacing every occurrence of a constant, each occurrence is treated separately. In some instances this lead to additional invariants.

2.1.3 Variable aging

When a variable is aged, it is replaced by the value that the variable had one iteration earlier. For example if we have a loop that in each execution performs the task *i := i + 1*, and *i* is not changed in any other way, then *aging(i)* would return the expression *i -1*.
The current implementation creates two mutations for each variable which needs to be aged; either by once decreasing the value by one, or by once incrementing the value by one. In practice this often achieves aging. A more sophisticated aging algorithm might be added to the algorithm at a later time. An example where this yields a correct result can be seen in the feature *max_v2_paper*, which is in the appendix.

3 Inferring loop invariants from postconditions: The algorithm and its implementation

This section demonstrates and explains the general algorithm as well as the implementation behind the loop invariant inference based on postcondition mutation.

The algorithm is adapted from [2] and implemented in Eiffel. The implementation relies on AutoProof's intermediate AST representation of the Eiffel code that needs to be analyzed. This approach has the advantage that AutoProof's existing functionalities can be used as an API, for example to check if a mutated postcondition is a valid invariant for some loop. All classes which contain the prefix *IV_* or *E2B_* in their name are part of AutoProof's implementation, while the *IV_* prefix specifically denotes classes used in AutoProof's intermediate representation. For consistency, classes added with this implementation also use the prefix when appropriate.

3.1 Main Algorithm

The main algorithm can be found in the class *IV_MUTATOR*. It is presented below as Code 1.

```
invariants():LINKED_LIST[IV_EXPRESSION]
        -- This is adapted from the paper "Inferring Loop Invariants using
           Post-conditions". It is called invariants there.
        -- It returns a list of expressions that are invariant of some loop
           in the procedure.
        local
            mutations: LINKED_LIST[IV_EXPRESSION]
            -- expressions in this list are mutated, but not yet tested if
               they are valid invariants.
            formula: IV_EXPRESSION
            any_loop: ARRAY[IV_BLOCK]
        do
            across postconditions as post loop
                across outer_loops as loops loop
                    -- compute all mutations of post
                    -- according to chosen strategies
                    mutations := mutate(post.item, loops.item)
                    across mutations as ms loop
                        formula:= ms.item
                        across all_loops as al loop
                            any_loop := al.item
                            if is_invariant(formula, any_loop) then
                                Result.force (formula)
                            end
                        end
                    end
                end
            end
        end
```

Code 1: The main algorithm to find loop invariants by mutating postconditions of a feature.

The 10th line of the code loops through all the postconditions of the feature that is currently being analyzed. The next line then loops through all outer loops of the feature. Since the targets of an inner loop are also targets of its encapsulating outer loop, it is enough to loop through outer loops when the mutating feature is called. In line 14, the mutations of the current postcondition are generated, using the feature which is presented in section 3.2. Lines 15 through 23 then loop through all the generated mutations. Each mutation is attached as an invariant to every loop and checked for its validity in line 19 of the code. If it passes this test, it is added to the *Result* list in the next line. How this test is performed is explained in section 3.2.3

3.2 Generating mutations of Postconditions

The code snippet Code 2 shows the overall process of how mutations are generated. The usage of this feature within the API is discussed in section 6.3.

This feature takes two arguments. The first is the postcondition which should be mutated. The second argument is an array representing a loop. It contains three blocks, the “head”, “body” and “end” of a loop. These three items are how loops are stored in AutoProof’s intermediate AST representation. The *Result* of this feature is a linked list of expressions. Each expression represents a mutation that is created based on heuristics. The reasoning behind the heuristics used in this implementation is further explained in section 2.

Lines 16 and 17 of Code 2 create the *Result*, which is a list of expressions, and add the postcondition itself as a first expression. Lines 18 through 23 are responsible for finding all basic sub-expressions of the postcondition. In the 24th line, all expressions which appear as a target of an assignment in the body of *a_loop* are stored in the list *targets_l*. The next two lines then fill the list *constant_l* with all sub-expressions which aren’t targets. So $\text{constant\_l} = \text{all\_subexpressions} / \text{targets\_l}$. Lines 27 and 28 then loop through the list of all constants. For each constant, the lines 29 through 37 then loop through all targets and perform coupled and uncoupled mutations, based on what options are selected in AutoProof’s window. These options are all presented in section 4. Line 39 of the code then removes all duplicate mutations from the *Result*. This is done because checking if a mutation is a valid invariant can be very time consuming, so it shouldn’t have to be done more often than necessary.

The code snippets for coupled and uncoupled mutations can be found in section 3.2.1.
The arguments of both features are the same: A postcondition, two expressions representing a constant and a variable and a loop. Both features also return the same type of result, namely a list of expressions, representing mutated post conditions.
The feature performing coupled mutations works as follows: Every occurrence of *a_constant* in the postcondtion *a_post* is replaced by *a_variable* in line 9 of the code. The remaining lines first

perform aging on *a_variable*, and then replace every occurrence of *a_constant* in the postcondition by the aged variables. What aging exactly does is explained in section 3.2.2.
The difference between the result list of uncoupled and coupled mutations is that *uncoupled_mutations* doesn't replace all occurrences of a constant. It rather replaces one occurrence at a time, and generates a new mutation for each constant which is replaced.

```
mutate(a_post:IV_POSTCONDITION; a_loop: ARRAY[IV_BLOCK]):
                                          LINKED_LIST[IV_EXPRESSION]
  --Adapted from paper "Inferring Loop Invariants using Postconditions".
  local
      all_subexpressions: LINKED_LIST[IV_EXPRESSION]
      bool_type: IV_BASIC_TYPE
      all_types: IV_TYPES
      targets_l: LINKED_LIST[IV_EXPRESSION]
          --list of all the targets in the loop
      constants_l: LINKED_LIST[IV_EXPRESSION]
          --list of all_subexpressions minus list of targets
      constant: IV_EXPRESSION
          --single item of constant_l
      variable: IV_EXPRESSION
          --same as in paper
  do
      create Result.make
      Result.force (a_post.expression)
      create all_subexpressions.make
      create all_types
      all_subexpressions.append (subexp (a_post.expression,
                                                all_types.bool))
      all_subexpressions.append (subexp (a_post.expression,
                                                all_types.int))
      all_subexpressions.append (subexp (a_post.expression,
                                                all_types.real))
      all_subexpressions.append (subexp (a_post.expression,
                                                all_types.heap_type))
      targets_l := targets (a_loop.item (2))
      create constants_l.make
      [..] -- fill constants_l with subexpressions which aren't targets.
      across constants_l as cs loop
          constant := cs.item
          across targets_l as ts loop
              variable := ts.item
              if options.is_coupled_mutations_enabled then
                  Result.append (coupled_mutations (a_post, constant,
                                                      variable, a_loop))
              end
              if options.is_uncoupled_mutations_enabled then
                  Result.append (uncoupled_mutations (a_post, constant,
                                                      variable, a_loop))
              end
          end
      end
      Result := remove_duplicates (Result)
  end
```

Code 2: The postcondition mutation algorithm

3.2.1 Generating coupled and uncoupled mutations

The actual implementations of both functions are shown in Code 3. *Coupled_mutations* first replaces every occurrence of *a_constant* in *a_post* with *a_variable* and adds this new expression to the *Result* list (line 9). In the next line the variable is then aged, which results in a list of new variables. For each one of these new variables the same replacement strategy is chosen again as in the beginning (lines 11-13).

As described in section 2.1, *uncoupled_mutations* has to replace each occurrence of the constant one by one, without changing the other occurrences. The loop which starts in line 25 replaces one occurrence after the other by *a_variable* and its mutations, each time generating a new expression and adding it to the *Result* list (lines 31-35). Once the generated expression is equal to the postcondition, we know that nothing has been replaced, and the loop can stop (lines 27-29).

```
coupled_mutations(a_post: IV_POSTCONDITION; a_constant, a_variable:
    IV_EXPRESSION;a_loop: ARRAY[IV_BLOCK]): LINKED_LIST[IV_EXPRESSION]
--Adapted from paper "Inferring Loop Invariants using Postconditions".
--a_loop is passed because it is needed when aging is called.
local
    aged_variable: IV_EXPRESSION
    all_aged_variables: LINKED_LIST[IV_EXPRESSION]
do
    create Result.make--line 5
    Result.force(replace_all (a_post, a_constant, a_variable))
    all_aged_variables := aging (a_variable, a_loop)
    across all_aged_variables as av loop
            Result.force (replace_all (a_post, a_constant, av.item))
    end
end

uncoupled_mutations(a_post: IV_POSTCONDITION; a_constant, a_variable:
   IV_EXPRESSION; a_loop: ARRAY[IV_BLOCK]): LINKED_LIST[IV_EXPRESSION]
local
    index: INTEGER
    expression_replaced: IV_EXPRESSION
        --the expression of a_post, with the i'th occurrence of
          a_constant replaced by a_variable (i'th = index)
    aged_variables: LINKED_LIST[IV_EXPRESSION]
do
    create Result.make
    index := 1
    from
        expression_replaced := replace_nth (a_post, a_constant,
                                                a_variable, index, a_loop)
    until
    --loop until nothing is replaced, so until replace_nth returns the
      same result
        expression_replaced.is_deep_equal(a_post.expression)
    loop
        expression_replaced := replace_nth (a_post, a_constant,
                                                a_variable, index, a_loop)
        Result.force (expression_replaced)
        aged_variables := aging (a_variable, a_loop)
        across aged_variables as av loop
            Result.force (replace_nth (a_post, a_constant, av.item,
                                                      index, a_loop))
        end
        index := index + 1
    end
end
```

Code 3: Implementations of coupled and uncoupled mutations.

3.2.2 Aging variables

```
    aging(a_variable: IV_EXPRESSION; a_loop: ARRAY[IV_BLOCK]):
LINKED_LIST[IV_EXPRESSION]
    2   local
    3       bin: IV_BINARY_OPERATION
    4       val: IV_VALUE
    5       t: IV_BASIC_TYPE
    6   do
    7       create Result.make
    8       if options.is_aging_enabled then
    9           if a_variable.type.is_integer then
    10              create t.make_integer
    11              create val.make ("1", t)
    12              create bin.make (a_variable.twin, "-", val, t)
    13              Result.force (bin)
    14              create bin.make (a_variable.twin, "+", val, t)
    15              Result.force (bin)
    16          end
    17      end
    18  end
```

Code 4:Implementation of the aging algorithm.

The aging feature performs a very simple task in the current implementation. If the value it receives as an expression is of type INTEGER, it creates two new expressions of type *IV_BINARY_EXPRESSION* (lines 12, 14). The first is the variable minus one; the second is the variable plus one. It then returns these two expressions in a linked list. The *if* statement in line 8 makes sure the code is only executed if the option is selected in the GUI.

3.2.3 Checking if a formula is an invariant of a loop

This feature first clones the universe so that the universe used by AutoProof is not changed and then attaches the formula as an invariant to the desired loop in the new universe (lines 14-16). Lines 17 through 25 then make use of AutoProof's API. Boogie code is generated based on the new universe and a verifier is called on this code. The result of the verifier is then stored in *res*. If the result shows that either no feature returned an error (line 27), or at least the feature which contains the loop with the new invariant returned no error (line 33), then we know that *a_formula* is an invariant of *a_loop*. Otherwise we have to inspect each error message separately (lines 38, 43). If the code of any error message indicates that the invariant failed, then *a_formula* is not considered to be an invariant of the loop (line 46).
The codes in line 45 are the ones which are generated in the feature *create_error* of the class *E2B_OUTPUT_PARSER*.

```
is_invariant(a_formula: IV_EXPRESSION; a_loop: ARRAY[IV_BLOCK]): BOOLEAN
    --returns True iff a_formula is an invariant of a_loop.
    local
        ver: E2B_VERIFIER
        gen: E2B_BOOGIE_GENERATOR
        ver_input: E2B_VERIFIER_INPUT
        new_universe: IV_UNIVERSE
        res: E2B_RESULT
        attacher: IV_ATTACH_INVARIANT
            --visitor to attach invariant to a_loop in the universe.
    do
        -- Attach invariant and launch Boogie verifier
        Result := False
        new_universe := universe.deep_twin
        create attacher.make (a_loop.item (1), a_formula)
        new_universe.process (attacher)
        create gen.make (new_universe)
        gen.generate_verifier_input
        ver_input := gen.last_generated_verifier_input
        ver := Void
        create ver.make
        ver.set_input (ver_input)
        ver.verify
        ver.parse_verification_output
        res := ver.last_result

        if res.failed_count = 0 and res.verified_count > 0 then
        --it must be an invariant, since no errors were detected anywhere.
          if both are 0 then something failed.
            Result := True
        else
        --find out in which procedure a_loop is, then check if that
        procedure is in the verified list. if so, set Result to True.
            across res.verified_procedures as verified_procs loop
                if (not (attacher.attached_to_procedure = Void)) and then
                    (attacher.attached_to_procedure.name.is_equal
                                (verified_procs.item.procedure_name)) then
                    Result := True
                end
            end
        end
        if (not Result) and not (res.failed_count = 0 and
                                      res.verified_count = 0) then
        -- Until now it was only added it if the whole feature didn't have
           any errors.
        -- So if anything couldn’t be proven, it wasn't added.
        -- Add check here to see if no invariant was found to be false.
            Result := true -set false once an error is found.
            across res.verification_errors as ver_errors loop
                across ver_errors.item.errors as errs loop
                    if errs.item.code ~ "BP5004" or errs.item.code ~
                    "BP5005" or errs.item.code ~ "BP5001loop_inv" then
                        Result := false
                    end
                end
            end
        end
    end
```

Code 5: Implementation of the feature which tests if a_formula is a valid invariant of a_loop with a verifier.

3.3 Implementation challenges

During the process of development of the code, several challenges came up. The main ones are presented here together with their solutions.

- While AutoProof's intermediate AST representation of the Eiffel code brings many advantages to the current implementation, it also presents some challenges. The AST is originally constructed to produce Boogie code from it. So once an invariant is found, it must be translated back to Eiffel Syntax, which isn't always trivial. For example correctly translating from Boogie's "forall" structure to Eiffel's equivalent, which is an "across" structure. An API to handle this translation is presented in section 6.1.
- The feature *is_invariant* calls a verifier to check if an expression is a valid invariant of a loop. If the feature which is being checked has no other errors, this decision is easy since the verifier returns no errors. If one or more errors are found, these errors must be inspected to see if they are relevant to the attached invariant. The current implementation looks at the codes of the error code, which are generated in the class *E2B_OUTPUT_PARSER*, to see if an invariant failed. This however requires that there are no other faulty invariants in the original Eiffel code which is being analyzed. It would be possible to remove all other invariants; however this then has the disadvantage that a removed invariant might be needed to prove the correctness of the current invariant, which would fail in this case.

4 Integration within AutoProof's UI and its usage

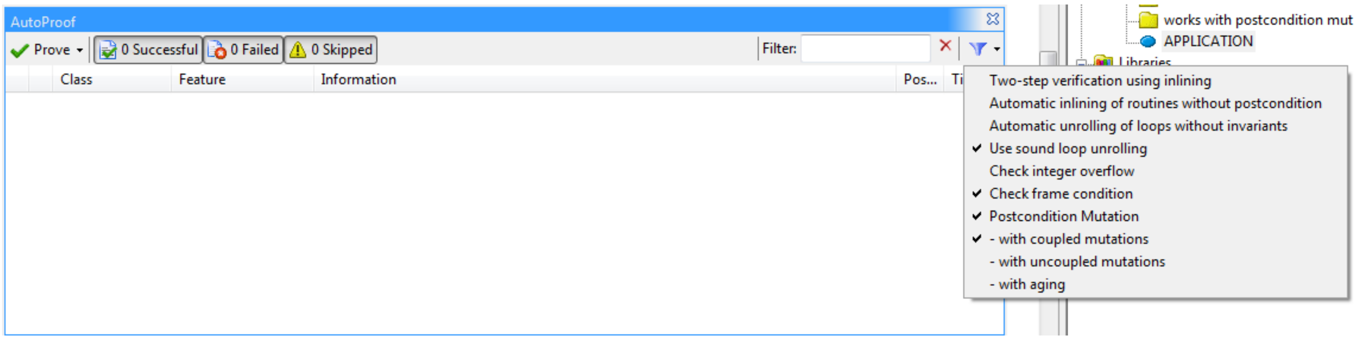
This section demonstrates how a developer can make use of the automatic invariant generation by postcondition mutation within EVE.

First, AutoProof's window needs to be opened. This can be done under View -> Tools -> AutoProof.

The usage of the postcondition mutation algorithm is integrated within AutoProof's window. In the options section, four options relevant to postcondition mutation can be set or disabled. The "Postcondition Mutation" option is the main entry. The postcondition mutation and invariant generation algorithm will only be executed if this option is set. If it is disabled, the algorithm will not be executed, regardless of whether the following three options are set:

- With coupled mutations
- With uncoupled mutations
- With aging.

These options define if their corresponding parts of the algorithm should be executed (see section 3). This means for example the coupled mutations will only be generated, if its option is set. The aging option is the only one of these three which is dependent on the other two. It will only generate more candidates if coupled mutations or uncoupled mutations (or both) is selected. Each part can potentially generate more invariant candidates. However, each candidate needs to be checked by Boogie which can increase the running time. This increase can be a few seconds up to a few minutes per candidate. Depending on which mutation options are selected, and on the postcondition which is currently being mutated, usually between one to about fifty candidates are generated (Of course there might be a lot more, for example if the postcondition has many replaceable sub expressions). It is therefore advised to first have fewer mutations generated, and increasing the number if no desired invariant is found. By default, only the options for postcondition mutation and coupled mutations are set, while the other two are disabled.



AutoProof

Prove | 0 Successful | 0 Failed | 0 Skipped | Filter:

	Class	Feature	Information	Pos...	Ti

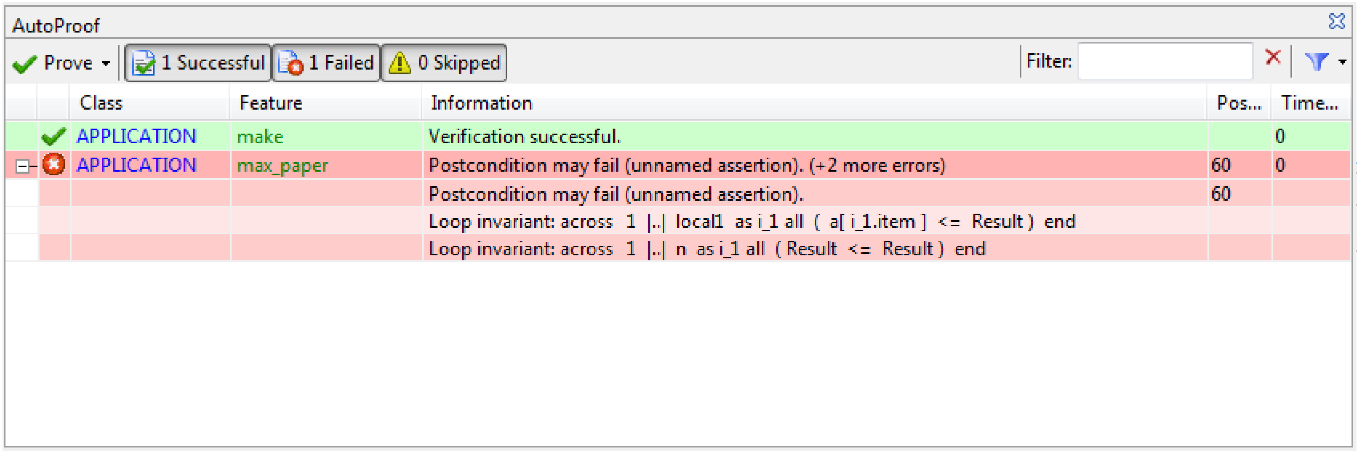
works with postcondition mut
APPLICATION
Libraries

- Two-step verification using inlining
- Automatic inlining of routines without postcondition
- Automatic unrolling of loops without invariants
- ✓ Use sound loop unrolling
- Check integer overflow
- ✓ Check frame condition
- ✓ Postcondition Mutation
- ✓ - with coupled mutations
- - with uncoupled mutations
- - with aging

Figure 1: The default selection of the options of the postcondition mutation algorithm within AutoProof's window.

```
feature


    max_Paper(a: ARRAY[INTEGER]; n: INTEGER):INTEGER
        require
            size_of_array_is_n: a.count = n
            n_positive: n>=1
        local
            i: INTEGER
        do
            from i:=0; Result := a[1];
            until i>= n
            loop
                i:= i+1
                if Result <= a[i] then
                    Result := a[i]
                end
            end
        ensure
            across 1 |..| n as j all a[j.item] <= Result end
            -- example of a postcondition that is trivially always true.
            -- across 1 |..| n as j all (n = n + n + a[j.item] - n - a[j.item] - 0) end
        end
```



AutoProof

Prove | 1 Successful | 1 Failed | 0 Skipped | Filter:

	Class	Feature	Information	Pos...	Time...
✓	APPLICATION	make	Verification successful.		0
⊟ ✗	APPLICATION	max_paper	Postcondition may fail (unnamed assertion). (+2 more errors)	60	0
			Postcondition may fail (unnamed assertion).	60	
			Loop invariant: across 1 |..| local1 as i_1 all (a[i_1.item] <= Result) end		
			Loop invariant: across 1 |..| n as i_1 all (Result <= Result) end		

Figure 2: Automatically generated loop invariants using coupled mutations and aging are presented within AutoProof's window. The term "local1" in the first generated invariant must be replaced by i before copying it into the Eiffel code as an invariant. Once the invariant is inserted, AutoProof is then also able to successfully verify the postcondition of this feature, so the whole feature is verified.

To execute the automatic generation of invariants, one can press the “Prove” button once all the wanted options are selected. EiffelStudio will then be unresponsive while all the selected tasks are performed. Once AutoProof is finished, it will display which features could or couldn’t successfully be verified. Alongside this information, each feature for which one or more loop invariant was generated can be expanded to display the generated invariant. These invariant then belong to some loop within the feature. They are in Eiffel’s syntax, so they can easily be copied into the code. One small adjustment still needs to be made, which is replacing terms of the form “locali”, where i is a number. The term “*local3*” for example would have to be replaced by the third local variable which was defined under the *local* clause of that feature.

5 Evaluation

Several experiments were performed to demonstrate the correct functionality of the implementation. The results are presented in **Table 1**.

Procedure	LOC	#	LP	M.V.	CND	INV	REL	T	SRC
max_paper	18	1	1	2	29	4	1	59	[2]
max_v2_paper	17	1	1	2	29	4	1	70	[2]
max_in_array	30	1	1	2	21+37+29	4	2	172	
add	20	1	1	2	9	1	1	20	
welfare_crook	33	1	1	3	49	25	20	107	[3]
seq_search_v1	30	1	1	3	13+13+49	11	6	136	[3]
seq_search_v2	27	1	1	1	25+17	16	1	86	[3]
dutch_flag	74	1	1	5	9+17+65+57+49	18	7	458	[3]
sum_and_max	30	1	1	3	7	1	1	22	

Table 1. *Performed experiments.*

The table should be read as follows:
LOC denotes how many lines of code the test cases contain. The total number of loops that an experiment contains can be found in the **#** coloumn, while the maximum amount of nested loops within a feature is denoted as **LP**. **M.V.** says how many variables are modified by the loop. **CND** denotes how many different mutations were created per postcondition, i.e. how many invariant candidates there were, while **INV** is the number of candidates that passed the test and are therefore valid invariants. The next coloumn, **REL**, then has the amount of these valid invariants, which are actually relevant to the code e.g. excluding tautological expressions. The second to last coloumn, **T**, denotes the running time that passed from starting the mutation in the GUI, until AutoProof finished. **SRC** refers to the source where the example was found. In some instances the examples are translated from other programming languages to Eiffel. The experiments which are adapted from [3] were originally written in Boogie processed with gin-pink. Their original results can be found in [2].
The experiments are executed on an Intel Core 2 Quad CPU Q9400 @2.66GHz running windows 7 64-bit operating system, using EVE based on EiffelStudio 7.2.0.0 GPL Edition. All options within the postcondition mutation algorithm were activated in each case. In most cases this isn't necessary to find the invariant.

Comments on some of the test cases:
max_paper: once the generated invariant is added, AutoProof can successfully prove the routine.
max_v2_paper: once the generated invariant is added, AutoProof can successfully prove the routine.
max_in_array: once BOTH relevant generated invariants are added, AutoProof can successfully prove the routine.
sum_and_max: AutoProof can only verify the generated invariant if the other two invariants in the code are provided.

In some cases, invariants were generated correctly based on the mutations, but were not presented to the developer because AutoProof could not verify the generated invariants to be correct, even if the invariants are manually inserted into the Eiffel code. These results are presented in **Table 2**.

Procedure	LOC	#	LP	M.V.	CND	INV	REL	T	SRC
mjrty	48	1	1	2	25+28	2	0	251	[3]
partition1	48	1	1	3	55	0	0	132	[3]

Table 2: *Test cases for which invariants were correctly generated based on the specifications of the postcondition mutation algorithm, but couldn't be verified due to AutoProof's limitations.*

6 Postcondition mutation API

The implementation offers several functions to be used in the form of an API. The useful classes and their features are presented in this section.

6.1 *IV_EXPRESSION_2_EIFFEL_POSTCONDITION*

This class is designed to take an *IV_EXPRESSION* and generate a string that represents this expression using Eiffel Syntax. The usage form is as follows:

```
feature

    converter(e: IV_EXPRESSION): STRING

        local
            printer: IV_EXPRESSION_2_EIFFEL_POSTCONDITION
        do
            create printer.make
            converter.process(printer)
            Result := printer.output
        end
```

Code 6: How to use an object of type IV_EXPRESSION_2_EIFFEL_POSTCONDITION. If printer is needed again later, printer.reset must be called first.

6.2 *IV_EXPRESSION_REPLACER*

This class is designed to replace a sub-expression of a given expression by a different sub-expression. The expression that should be processed must be of type {*IV_EXPRESSION*}. The input expression itself will not be changed, only analyzed and copied, and a new output expression is generated which has the correct expressions replaced. The interesting functions are presented in the following subsections.

6.2.1 Creation procedure

```
    make_nth(a_post_expression: IV_EXPRESSION; a_old, a_new: IV_EXPRESSION;
                                                        a_n_th: INTEGER)
        -- replace the a_n_th occurence of a_old in a_post with a_new

        -- if a_n_th is <=0, replace all occurences of a_old
```

To create an object of type *IV_EXPRESSION_REPLACER*, this must be called. *a_post_expression* is the main expression, of which sub-expressions should be replaced. *a_old* is the sub-expression which should be replaced by the new sub-expression, *a_new*. If *a_n_th* is set to be less or equal to zero, all occurrences of a_old in a_post_expression will be replaced by *a_new*. Otherwise only the *a_n_th* occurrence will be replaced.

6.2.2 Output

Once this object is created, it can be used to process an object of type {*IV_EXPRESSION*} since it inherits from {*IV_EXPRESSION_VISITOR*}. The newly generated expression with the replaced sub-expressions is then stored in *output*, where it can be accessed.

6.2.3 Example

An example of how to use an object of this type is presented in the following code block.

```
    replace_all(a_post: IV_POSTCONDITION; a_old, a_new: IV_EXPRESSION):
IV_EXPRESSION
        -- replace every ocurrence of a_old in a_post with a_new
        -- uses class IV_EXPRESSION_REPLACER
        local
            replacer: IV_EXPRESSION_REPLACER
        do
            create replacer.make_nth (a_post.expression, a_old, a_new, 0)
            a_post.expression.process (replacer)
            Result := replacer.output
        end
```

Code 7 shows an example of how to use the class IV_EXPRESSION_REPLACER. This feature is taken from the class IV_MUTATOR.

6.3 IV_MUTATOR

This is the class where the main algorithm lies and most of the computation is performed. A detailed explanation was given in section 3. The interesting features that can be used as part of the API are presented here.

6.3.1 Creation procedure

```
make (a_universe: IV_UNIVERSE; a_options: E2B_OPTIONS)
```

When creating an object of this type, this feature must be called. The first argument delivers the intermediate AST representation that should be analyzed. The second argument is used to check which parts of the algorithm should be executed. If it should be different options than what are selected in the GUI, a new object can be created. The flags in the class *E2B_OPTIONS* that are relevant and must be set accordingly are:

- *is_postcondition_mutation_enabled*
- *is_coupled_mutations_enabled*
- *is_aging_enabled*
- *is_uncoupled_mutations_enabled*

By default the first two are set to true, while the other two are set to false in the creation feature.

In the current implementation all the arguments are generated by AutoProof and passed on in the creation procedure of the class *E2B_VERIFY_TASK*.

6.3.2 Processing an implementation which contains a loop and postcondition

```
process_implementation (a_implementation: IV_IMPLEMENTATION)
            -- Process implementation `a_implementation'.
            -- this should be the first feature called after creation.
```

This is the feature that executes the main algorithm. It should usually be called right after the creation procedure. After this is executed, all the generated invariants are stored in the *output* feature. The argument should be an object that represents an implemented feature and is within the universe. The feature it represents should also contain one or more loops and postconditions. Executing this feature might have a long running time, since several invariant candidates might be generated and checked with Boogie, similar to executing the algorithm from within the GUI.

6.3.3 Generating all mutations

```
mutate(a_post:IV_POSTCONDITION; a_loop: ARRAY[IV_BLOCK]):
                                          LINKED_LIST[IV_EXPRESSION]
```

This feature is automatically called within *process_implementation*, but can also be called separately later on. It is used to generate mutations of the postcondition that is passed as the first argument for the loop that is passed as the second argument. The *Result* contains all the mutated expressions. These have not yet been checked if they are actually invariants of the loop.

6.3.4 Checking if an expression is a valid loop invariant

```
is_invariant(a_formula: IV_EXPRESSION; a_loop: ARRAY[IV_BLOCK]): BOOLEAN
        --returns True iff a_formula is an invariant of a_loop.
```

This feature returns true if and only if *a_formula* is an invariant of the loop represented by the second argument. Calling it can be very time consuming, since Boogie code is generated and checked with a verifier.

6.3.5 Generating coupled and uncoupled mutations

The signatures of the features are as follows:

```
coupled_mutations(a_post: IV_POSTCONDITION; a_constant, a_variable:
      IV_EXPRESSION;a_loop: ARRAY[IV_BLOCK]): LINKED_LIST[IV_EXPRESSION]
uncoupled_mutations(a_post: IV_POSTCONDITION; a_constant, a_variable:
      IV_EXPRESSION; a_loop: ARRAY[IV_BLOCK]): LINKED_LIST[IV_EXPRESSION]
```

These features are used to generate coupled and uncoupled mutations of the expression of *a_post*. The Result is a linked list of all mutated expressions.

6.3.6 Extracting sub-expressions

```
subExp(an_expression: IV_EXPRESSION; a_type:IV_TYPE):
                                                LINKED_LIST[IV_EXPRESSION]
```

Returns a list of expressions that are sub-expressions of *an_expression* and of type *a_type*.

6.3.7 Other features

Various features which are accessible within the same package are presented here. These can all be used once the creation procedure and the feature *process_implementation* both have been called.

```
body: IV_BLOCK
        --body of the implementation
contracts: LINKED_LIST [IV_CONTRACT]
        --all the contracts of this implementation.
postconditions: LINKED_LIST [IV_POSTCONDITION]
        --list of copy of all the postconditions of this implementation.
outer_loops: LINKED_LIST [ARRAY[IV_BLOCK]]
        --list of outer loops in the procedure. Each loop is saved as an array
          of 3 items, loop_head_X, loop_body_(X+1), loop_end_(X+2)
all_loops: LINKED_LIST [ARRAY[IV_BLOCK]]
        --list of all loops (inner and outer) in the procedure. Each loop is
          saved as an array of 3 items, loop_head_X, loop_body_(X+1),
          loop_end_(X+2)
universe: IV_UNIVERSE
        --the complete universe that the AST for this implementation was taken
          from.
```

6.4 *IV_POSTCONDITION_MUTATION*

This class is built as a visitor and inherits from both *IV_UNIVERSE_VISITOR* and *IV_STATEMENT_VISITOR*. It visits the main nodes of the universe, and once an implementation node is found which contains a loop, it calls an object of type *IV_MUTATOR* on it to generate loop invariants. Once this is done, this class is also responsible for displaying the found invariants in AutoProof's window, as seen in the following subsection.

6.4.1 Displaying an invariant for the user

```
display_invariant(a_implementation: IV_IMPLEMENTATION; a_output:
LIST[IV_EXPRESSION])
        --displays the generated invariants on the screen, in AutoProofs
          window. a_implementation is the implementation of the procedure that
          the inviariant belongs to.
```

This feature is responsible for displaying the generated loop invariables within AutoProof's windows. It does so by adding the generated invariants to the object of type *E2B_VERIFIER*, which is used by AutoProof to display the results. If the results should be displayed in a different manner, this feature can be adjusted accordingly.

7 Conclusions

The current implementation fulfills the goal of integrating an automatic invariant inference from postcondition mutations algorithm by using the techniques presented in [2]. A developer using EVE can launch the procedure within AutoProof's GUI without having any further knowledge of how the inference works, by just selecting all the options. A more knowledgeable developer might leave certain options unchecked, so that the process might be sped up by not creating as many mutations, and later using other options if no invariant was found.

Furthermore an API is presented in section 6 which offers some additional functionality to those available in the GUI. This allows the functionality of the current implementation to be used and extended later on.

The main **challenges** which were met during development are:

- Understanding how AutoProof's intermediate AST representation is built up. Once this learning process was done, the additional functionality that was gained was very helpful and sped up the overall implementation time.
- Finding a way to translate the intermediate representation back to Eiffel syntax, so it can be presented to the developer.

Several non-trivial ways in which the current implementation might be **enhanced** or altered are:

- The aging algorithm could be optimized to actually find the value a variable had in the last execution of the body.
- A generated invariant might be a tautology, in which cases the current implementation presents it to the developer as an invariant. The developer should then notice this and not add it as an invariant, since it doesn't help. However, even if it is added, it shouldn't affect any proof of correctness of the program.
- Since Boogie isn't complete in the sense that it cannot always find a proof even if there is one, some invariants might not be presented to the developer although they are generated correctly. The implementation could differentiate between invariants which are proven to be false, and those for which Boogie can find no proof, and present the latter to the developer to make the final decision.
- If there is a 'check' instruction directly after a loop, this function could be treated as the loop's postcondition and be mutated together with the feature postconditions.

8 Appendix

8.1 Source code of the test cases that were used for evaluation

All the source codes which were used during the evaluation are presented here. Invariants which are in the code are needed for AutoProof to be able to verify the procedures. Some features are commented out and replaced by equivalent inline postconditions to enable AutoProof to verify the feature.

```
max_Paper(a: ARRAY[INTEGER]; n: INTEGER):INTEGER
    require
        size_of_array_is_n: a.count = n
        n_positive: n>=1
    local
        i: INTEGER
    do
        from i:=0; Result := a[1];
        until i>= n
        loop
            i:= i+1
            if Result <= a[i] then
                Result := a[i]
            end
        end
    ensure
        across 1 |..| n as j all a[j.item] <= Result end

    end

    max_v2_Paper(a: ARRAY[INTEGER]; n: INTEGER):INTEGER
    require
        size_of_array_is_n: a.count = n
        n_positive: n>=1
    local
        i: INTEGER
    do
        from i:=1; Result := a[1];
        until i> n
        loop
            if Result <= a[i] then
                Result := a[i]
            end
            i:= i+1
        end
    ensure
        across 1 |..| n as j all a[j.item] <= Result end
    end
```

```
max_in_array (a: ARRAY [INTEGER]): INTEGER
        note
            pure: True
        require
            a /= Void
            a.count > 0
        local
            x, y: INTEGER
        do
            from
                x := 1
                y := a.count
            invariant
                y >= x
                across 1 |..| x as i all a[i.item] <= a[x] or a[i.item] <=
                                                                  a[y] end
                across y |..| a.count as i all a[i.item] <= a[x] or a[i.item]
                                                                  <= a[y] end
            until
                x = y
            loop
                if a[x] <= a[y] then
                    x := x + 1
                else
                    y := y - 1
                end
            end
            Result := x
        ensure
            1 <= Result and Result <= a.count
            across 1 |..| a.count as i all a[i.item] <= a[Result] end
            across a as i all i.item <= a[Result] end
        end
```

```
add (a, b: INTEGER): INTEGER
    require
        a > 0
        b > 0
    local
        i: INTEGER
    do
        from
            Result := a
            i := 0
        invariant
            Result = a + i
        until
            i >= b
        loop
            Result := Result + 1
            i := i + 1
        end
    ensure
        Result = a + b
    end
```

```
        p_f, p_g, p_h: INTEGER
        find_crook(left:INTEGER; F,G,H: ARRAY[INTEGER])
            require
                left >= 1
                left <= F.count
                left <= G.count
                left <= H.count
                F /= Void
                G /= Void
                H /= Void
            local

            do
                from p_f := left-1;
                p_g := left-1;
                p_h := left-1;
                until (not (F[p_f+1] /= G[p_g+1]  or  G[p_g+1] /= H[p_h+1]))
                loop
                    if F[p_f+1] < G[p_g+1] then
                        p_f := p_f + 1;
                    else
                        if G[p_g+1] < H[p_h+1] then
                            p_g := p_g + 1;
                        else
--                          check H[p_h+1] < F[p_f+1] end
                            p_h := p_h + 1;
                        end

                    end
                end
            ensure
                en1: p_f+1 >= left and p_g+1 >= left and p_h+1 >= left;
            end
            --return values
            found: BOOLEAN
            p: INTEGER
            seq_search (a: ARRAY[INTEGER]; n: INTEGER; v: INTEGER)
                require
                    n>=0
                local
                    i: INTEGER
                do
                    i:= 1
                    found:= False
                    from
                    invariant
                    until not (i <= n  and  (not found))
                    loop
                        if a[i]=v then
                            p:=i
                            found := True
                        else
                            i := i + 1
                        end
                    end
                ensure
                    (not found) or (a[p] = v)
                    (not found) or (1 <= p and p <= n)
--              found or not_exists (v, a, 1, n)
                    found or across 1 |..| n as j all a[j.item] /= v end
                end
--      not_exists(v: INTEGER; a: ARRAY[INTEGER]; low, high: INTEGER): BOOLEAN
--      do
--          Result := across low |..| high as j all a[j.item] /= v end
--      end
```

```
        found: BOOLEAN
        p: INTEGER
        seq_search_v2 (a: ARRAY[INTEGER]; n: INTEGER; v: INTEGER)
            require
                n>=0
                n <= a.count
            local
                i: INTEGER
            do

                found:= False
                from i:= 1
                invariant
                    true
--                  not not across 1 |..| n as i25 all a[i - 1] /= v end
                until i > n  or else  ( A[i] = v)
                loop
                    i := i + 1;
                end
                if  i <= n  then
                    p := i;
                    found := true;
                end

            ensure
                not (not (across 1 |..| n as j all a[j.item] /= v end)) or
                  (found and a[p] = v) -- not exists, or found
                (not (across 1 |..| n as j all a[j.item] /= v end)) or (not
                  found) --exists or not found.
            end
```

```
--colors defined as 0: blue, 1:white, 2:red.
is_flag_color(i:INTEGER):BOOLEAN
    do
        Result := i = 0 or i = 1 or i = 2
    end
b:INTEGER
r:INTEGER
--invariant might not be maintained.
make_flag(a:ARRAY[INTEGER]; n: INTEGER): ARRAY[INTEGER]
    require
        n>=1
        n<=a.count
--      is_flag_color_array(a,1,n)
        across 1 |..| n as curr all is_flag_color(a[curr.item]) end
    local
        i: INTEGER
        tmp: INTEGER --used for swap
    do
        Result:= a
        from
            b:=1
            i:=1
            r:= n+1
        invariant
--          across 1 |..| (b-1) as curr all Result[curr.item] = 0 end
--          across b |..| (r-1) as curr all Result[curr.item] = 1 end
--          across r |..| n as curr all Result[curr.item] = 2 end
        until
            i>=r
        loop
            if (Result).item(i)=0 then
                --swap
                tmp:=(Result).item(i)
                (Result).item(i):= (Result).item(b)
                (Result).item(b)  := tmp
                -----
                i:= i+1
                b:= b+1
            else
                if (Result).item(i)  = 1 then
                    i  := i + 1
                else
                    r:= r-1
                    --swap
                    tmp:=(Result).item(i)
                    (Result).item(i):= (Result).item(r)
                    (Result).item(r)  := tmp
                    -----
                end
            end
        end
    ensure
        min_b: 1<=b
        b_min_r: b<=r
        r_min_n: r<=n+1
        blue: across 1 |..| (b-1) as curr all Result[curr.item] = 0 end
        white: across b |..| (r-1) as curr all Result[curr.item] = 1 end
        red: across r |..| n as curr all Result[curr.item] = 2 end
```

```
--      monochrome (Result, 1, b-1, 0)
--      monochrome (Result, b, r-1, 1)
--      monochrome (Result, r, n, 2)
    end
--  --gives semantic error in output.bpl
--  monochrome(a: ARRAY[INTEGER]; low:INTEGER; high: INTEGER; col:
INTEGER):BOOLEAN
--      require
--          is_flag_color(col)--needed?
--      do
--          Result:= across low |..| high as curr all a[curr.item] = col end
--      end

----gives semantic error in output.bpl
--is_flag_color_array(a:ARRAY[INTEGER]; low: INTEGER; high: INTEGER): BOOLEAN

--  do
--      Result:= across low |..| high as curr all is_flag_color(a[curr.item])
end
--  end
-- invariants were generated correctly, but could not be verified by Boogie,
even if they were inserted into the code.
-- only two tautologies were verified
cnt (c:INTEGER; a: ARRAY[INTEGER]; min: INTEGER; max: INTEGER):INTEGER
    require
        1 <= min and min <= a.count
        1 <= max and max <= a.count
    local
        i: INTEGER
    do
        from
        Result := 0
        i:= min
        until i > max
        loop
            if a[i] = c then
                Result := Result + 1
            end
            i:= i+1
        end
    end
cand: INTEGER
count: INTEGER
mjrty(a: ARRAY[INTEGER]; n: INTEGER)
    require
        n >= 0;
    local
        i,k: INTEGER
    do
        i := 0
        count := 0
        from
        until i >= n
        loop
            i:= i+1
            if count = 0 then
                count := 1;
                cand := A[i];
            else
                if A[i] = cand then
                    count := count + 1;
                else
                    count := count - 1;
                end
            end
        end
    ensure
        across a as all_els all (cnt(all_els.item,a,1,n) <= cnt(cand,a,1,n))
end
        --exists...:
        (across a as all_els all 2*cnt(all_els.item, a, 1, n) <= n end) or
(2*cnt(cand, a, 1, n) > n)
    end
```

```
-- invariant correctly generated but AutoProof cannot verify it, even if it is
added to code.
    index: INTEGER
    partition1 (a: ARRAY[INTEGER]; left : INTEGER; right: INTEGER; pivot:
INTEGER): ARRAY [INTEGER]
        -- partitions TODO
        require
            left <= right
        local
            i: INTEGER
            temp: INTEGER
        do
            Result := a
            i := left
            index := left

            from
            until i > right
            loop
                if Result[i] < pivot then
                    temp := Result[i] --swap
                    Result[i]:=Result[index]
                    Result[index]:= temp
                    index := index + 1
                end
                i := i + 1
            end

        ensure
--          across left |..| min (index, right)  as k all  Result[k.item] <=
                                pivot end -- doesnt work
            across left |..| index  as k all (left > right) or  Result[k.item]
                                            <= pivot end --doesnt work
--          across left |..| (index - 1) as k all a[k.item] < pivot end --this
            one works..
            -- is_LT_pivot (pivot, Result, left, right, index)
        end

    min(a,b:INTEGER):INTEGER
        do
            if a<b then
                Result := a
            else
                Result := b
            end
        ensure
            Result <= a and Result <= b
        end

    -- is_LT_pivot (pivot: INTEGER; a: ARRAY[INTEGER]; left: INTEGER; right:
      INTEGER;index_l: INTEGER): BOOLEAN
        -- require
            -- left <= right --TODO adjust this.
        -- do
            -- Result := across left |..| (index_l - 1) as k all a[k.item] <
                  pivot end
        -- end
    sum, max: INTEGER

     sum_and_max (a: ARRAY [INTEGER])
         note
             framing: False
         require
             a /= Void
             a.count > 0
             across a as ai all ai.item >= 0 end
         local
             i: INTEGER
         do
             from --loop_head_6
                 i := 1
                 sum := 0
             invariant
--               1 <= i and i <= a.count + 1
                across 1 |..| (i-1) as ai all a[ai.item] <= max end
                across a as ai all ai.item >= 0 end
--               sum <= (i-1) * max
             until
                 i > a.count
             loop
                 sum := sum + a[i]
                 if a[i] > max then
                     max := a[i]
                 end
                 i := i + 1
--           variant
--               a.count - i + 1
             end
--           check sum <= a.count * max end
         ensure
             sum <= a.count * max
         end
```

9 References

[1] "EVE," [Online]. Available: https://trac.inf.ethz.ch/trac/meyer/eve. [Accessed 28 June 2013].

[2] C. A. Furia and B. Meyer, *Inferring Loop Invariants Using Postconditions In Fields of Logic and Computation: Essays Dedicated to Yuri Gurevich on the Occasion of His 70thBirthday. Lecture Notes in Computer Science, 6300:277-300,* Springer, 2010.

[3] "gin-pink," [Online]. Available: http://se.inf.ethz.ch/people/furia/software/gin-pink.html. [Accessed 28 June 2013].

[4] "Boogie," [Online]. Available: http://research.microsoft.com/en-us/projects/boogie/. [Accessed 28 June 2013].

[5] J. Tschannen, C. A. Furia, M. Nordio and B. Meyer, "Usable Verification of Object-Oriented Programs by Combining Static and Dynamic Techniques," 2011.

[6] J. Tschannen, "Automatic verification of Eiffel programs. Master's Thesis.," ETH, 2009.