AutoProof Tutorial

Chair of Software Engineering, ETH Zurich

March 2015

Introduction

AutoProof is an auto-active[1] verifier for the Eiffel programming language; it proves functional correctness of Eiffel programs annotated with contracts. The goal of this tutorial is to show how to verify Eiffel programs with AutoProof through hands-on exercises.

Preparations

To use AutoProof locally you can install EVE on your machine. Although it is possible to use the online version of AutoProof to do the verification exercises, some options are not available on the web.

You can download the EVE delivery at

`http://se.inf.ethz.ch/research/eve/builds`

Downloads for the examples and exercises as well as links to using the web interface of AutoProof are available here:

`http://se.inf.ethz.ch/research/autoproof/tutorial`

Structure

Each of the following sections describes in detail the use of AutoProof based on increasingly complex examples. Each example is used throughout one section to explain some of the concepts behind AutoProof and how they are used to verify programs. Each section also has hands-on exercises with verification tasks for one or more programs.

[1] AutoProof tries to achieve an intermediate degree of automation in the continuum that goes from *auto*matic to inter*active*.

Contents

1 Verification of Basic Properties

To prove functional correctness automatically, a program needs a machine-readable specification. We are using Eiffel—an object-oriented programming language— which allows one to write contracts as part of the program. Each Eiffel routine is equipped with pre- and postconditions and each class has a class invariant.

We will use the *Account* example to show the basic concepts of AutoProof. The example consists of the two classes ACCOUNT and ACCOUNT_TEST. The first class models a bank account and the second class consists of two test cases that show proper and improper usage of the class. The full code of class ACCOUNT is shown in Figure 1.

First you can look through the example and verify the two classes; all routines, except for the deliberately failing test case, will be successfully verified.

1.1 Input Language

Eiffel Programs and Contracts

Here we give a short overview of the Eiffel programming language based on the *Account* example.

Class definition Eiffel is an object-oriented programming language. Classes are defined with the **class** keyword. If no inheritance clause is given (as in this example), then the class implicitly inherits from the class ANY, which serves as a common ancestor for all classes.

```
class
  ACCOUNT
```

Constructors To define constructors for the class, you can use the **create** keyword, followed by a comma-separated list of constructors called *creation routines*. If no constructor is defined, the routine default_create will implicitly become the only creation routine. In our example the routine make will be the creation procedure.

```
create
  make
```

```
note
  description: "Account class."
  model: balance, credit_limit

class ACCOUNT
create make

feature {NONE} -- Initialization

  make
      -- Initialize empty account.
    note
      status: creator
    do
      balance := 0
      credit_limit := 0
    ensure
      balance_set: balance = 0
      credit_limit_set: credit_limit = 0
    end

feature -- Access

  balance: INTEGER
      -- Balance of this account.

  credit_limit: INTEGER
      -- Credit limit of this account.

  available_amount: INTEGER
      -- Amount available on this account.
    note status: functional
    do
      Result := balance + credit_limit
    end

feature -- Element change

  set_credit_limit (limit: INTEGER)
      -- Set 'credit_limit' to 'limit'.
    require
      limit_valid: limit ≥ (0).max(−balance)
      modify_model ("credit_limit", Current)
    do
      credit_limit := limit
    ensure
      credit_limit_set: credit_limit = limit
    end

  deposit (amount: INTEGER)
      -- Deposit 'amount' in this account.
    require
      amount_non_negative: amount ≥ 0
      modify_model ("balance", Current)
    do
      balance := balance + amount
    ensure
      balance_set: balance = old balance + amount
    end

  withdraw (amount: INTEGER)
      -- Withdraw 'amount' from this account.
    require
      amount_not_negative: amount ≥ 0
      amount_available: amount ≤ available_amount
      modify_field (["balance", "closed"], Current)
    do
      balance := balance − amount
    ensure
      balance_set: balance = old balance − amount
    end

feature -- Basic operations

  transfer (amount: INTEGER; other: ACCOUNT)
      -- Transfer 'amount' from this to 'other'.
    require
      amount_not_negative: amount ≥ 0
      amount_available: amount ≤ available_amount
      no_aliasing: other ≠ Current
      modify (Current, other)
    do
      balance := balance − amount
      other.deposit (amount)
    ensure
      balance = old balance − amount
      other.balance = old other.balance + amount
      credit_limit = old credit_limit
      other.credit_limit = old other.credit_limit
    end

invariant
  limit_not_negative: credit_limit ≥ 0
  balance_not_credit: balance ≥ −credit_limit

end
```

Figure 1: Account example.

Features and visibility Routines and attributes (together called *features*) are defined in feature blocks using the **`feature`** keyword. Feature blocks can declare a visibility restriction by indicating a list of class names in curly braces. For example the first feature block restricts the access of the make routine to `NONE`, essentially hiding the routine from all other classes (no class can inherit from `NONE`). The other feature blocks do not have any access restriction and thus the features inside these feature blocks are public. It is common to name feature clauses by adding a comment using the double-dash `--` comment style (there are no multi-line comments in Eiffel).

```
feature {NONE} -- Initialization
feature -- Access
feature -- Basic operations
```

Attributes Attributes are defined with an attribute name followed by the type of the attribute. For all features it is common to add a comment on the line following the feature declaration.

```
  balance: INTEGER
    -- Balance of this account.
```

Routines A routine declaration consists of the routine name, optional parameters, optional return type, optional precondition, routine body and optional postcondition. The precondition denoted by the **`require`** keyword and postcondition denoted by the **`ensure`** keyword are the specification of the routine. The precondition holds prior to the execution of the routine, and the postcondition holds afterwards. Therefore the precondition is the responsibility of the client of the routine, whereas the postcondition has to be established by the routine itself. If a pre- or postcondition is omitted, the routine will have an implicit pre- or postcondition of **`True`**.

```
  set_credit_limit (limit: INTEGER)
      -- Set 'credit_limit' to 'limit'.
    require
      limit_valid: limit ≥ (0).max(−balance)
      modify_model ("credit_limit", Current)
    do
      credit_limit := limit
    ensure
      credit_limit_set: credit_limit = limit
    end
```

Assertion tags Each assertion, be it a precondition, a postcondition, a class or loop invariant, or an intermediate check instruction, can have an *assertion tag*. These tags are useful for debugging, as the feedback from AutoProof will specify the tag of violated assertions.

Class invariants Class invariants are written at the end of a class using the **invariant** keyword. Class invariants define the state of a consistent object and hold by default whenever an object is visible to other classes, for example at the beginning and end of each public routine.

```
invariant
  credit_limit_not_negative: credit_limit ≥ 0
  balance_not_below_credit: balance ≥ −credit_limit
```

There are more details on how to write an Eiffel program and what specification can be written for the verification with AutoProof; this will be explained throughout the rest of the tutorial.

AutoProof Annotations

AutoProof supports tow forms of custom annotations: note clauses for features and classes, and dummy routines made available through ANY.

Note clauses are used to denote special types of routines and attributes that influence the verification like creation routines (see Section 1.5) or ghost features (see Section 2.6). Additionally, note clauses are used to disable defaults for implicit pre-/postconditions of the ownership methodology (see Section 3.4).

The second form of AutoProof annotations are dummy features (routines and functions with empty implementation) that can be used in assertions or regular code. These features are defined in class ANY and are available everywhere. AutoProof gives special semantics to these features, for example to specify modifies clauses (see Section 1.4).

You can look at the AutoProof manual for a complete listing of custom annotations of both note clauses and dummy features[2].

1.2 Basic Properties

Booleans

The Eiffel boolean operations **not**, **and**, **or**, **xor**, and **implies** are supported by AutoProof. The semi-strict operators **and then** and **or else** are also

[2] http://se.inf.ethz.ch/research/autoproof/manual/#annotations

supported with the correct semantics that the right-hand side only needs to be valid if the left-hand side does not already define the overall value of the expression.

Integers

The Eiffel integer operations $+$, $-$, $*$, // (quotient of integer division), and \\ (remainder of integer division) are supported by AutoProof. Integers in AutoProof can be modeled in two modes, either as mathematical integers or as machine integers. By default integers will be modeled as mathematical integers, though AutoProof can also check overflows of bounded integers (see Section 2.8).

The Eiffel comparison operations on integers $=$, $\neq$, $<$, $>$, $\leq$, and $\geq$ are all supported.

References

Comparison of objects always uses reference equality. The standard equality operator `a = b` and inequality operator `a ≠ b` work as expected; object equality `a ∼ b` and inequality `a ≁ b` are not supported and will fall back to reference equality when used.

1.3 Models

AutoProof supports *model-based contracts*. Models are used to express the *abstract state space* of a class and describe its changes. To define the model of a class you add a ***model*** annotation to the **note** clause of the class. The model may only consist of attributes of the class.

```
note
  model: balance, credit_limit
class ACCOUNT ···
```

This makes the two attributes `balance` and `credit_limit` model fields of the class.

The idea behind model-based contracts is to have an abstract and concise yet expressive way to specify the interface of a class. When using models you use the class invariant to describe object validity in terms of the model attributes. The effect of each procedure is expressed by relating the pre-state of the model fields to their post-state. In addition you can express the framing specification in terms of the model fields.

The Mathematical Model Library (MML, see Section 2.1) can be used to model complex behavior. Also, ghost attributes might be introduced to define abstract behavior in terms of other functions or attributes and can then be used as model fields (see Section 2.6).

1.4 Framing

The framing model that AutoProof uses is based on *modifies clauses*. The `ACCOUNT` class deliberately used three different ways of specifying the modifies clause to demonstrate the differences between them.

`modify_model` (**`fields`**, **`objects`**)

Using *`modify_model`* you can specify that model fields may change during the execution of a routine. You can specify one or more model fields by providing as first parameter a manifest string with the name of the model attribute or a manifest tuple with multiple manifest strings. The second parameter is either a single object, a single set of objects of type `MML_SET`, or a manifest tuple with mixed objects or sets of objects.

```
deposit (amount: INTEGER)
  require
    ...
    modify_model ("balance", Current)
  do ··· end
```

This routine is allowed to modify the model field `balance` of the **`Current`** object.

The effect of *`modify_model`* is as follows: each model attribute specified in the *`modify_model`* clause *as well as each non-model attribute* can be modified in the routine. All model fields that are not listed remain unchanged. This means in turn that for clients all non-model attributes are potentially modified even though they are not listed in the modifies clause.

`modify_field` (**`fields`**, **`objects`**)

With *`modify_field`* you specify directly which attributes may be changed by a routine. As before, you can specify one or more attribute names by providing as first parameter a manifest string with the name of the model attribute or a manifest tuple with multiple manifest strings. The second parameter is again either a single object, a single set of objects of type `MML_SET`, or a manifest tuple with mixed objects or sets of objects.

This way of specifying the modifies clause is lower-level than specifying which model fields may change. This is also the reason we are required to add the ghost field `closed` in the example shown here. The `closed` field is

```
withdraw (amount: INTEGER)
  require
    ...
    modify_field (["balance", "closed"], Current)
  do ··· end
```

This routine is allowed to modify the attributes `balance` and `closed` of the `Current` object.

a boolean flag that is `True` whenever an object is in a consistent state (see Section 3 for details).

modify (objects)

The third option to specify modifies clauses is to give a list of objects which can be modified without limiting the modifications to certain attributes or model fields. For this modifies clause you can specify mixed objects or sets of objects.

```
transfer (amount: INTEGER; other: ACCOUNT)
  require
    ...
    modify (Current, other)
  do ··· end
```

This routine is allowed to modify all attributes of `Current` and `other`.

Since the objects may be modified freely, you have to specify the full effect on the modified objects. For example the `transfer` procedure of the account example, the postcondition not only describes the effect on the `balance` attribute of the two objects but also has clauses to specify that the `credit_limit` attribute does not change. This is for demonstration purposes only, it would be a better design to use *modify_model* instead (*try to change it!*).

Giving an empty tuple as argument—*modify* ([])—denotes that nothing may be modified, i.e., that the routine is *pure*.

Default Modifies Clauses

When no modifies clause is given a default modifies clause is used based on the type of routine:

- For **procedures** (routines without a return value), the default modifies clause is *modify* (`Current`). So all attributes can be modified in a procedure if no specific modifies clause is given.

- For **functions** (routines with a return value), the default modifies clause is *modify* ([]). Therefore, by default, all functions are *pure*.

When you overwrite the default modifies clause for procedures, for example to modify an object passed as parameter, and you want to be able to modify the **Current** object as well, you will need to add *modify* (**Current**) to the modifies clause (or a more specific version when only a subset of the attributes needs to be modifiable).

Combining modifies annotations

You can add several modifies annotations to a modifies clause. The set of modifiable objects and attributes is the union of all modifies annotations.

1.5 Routine Annotations

Creation Procedures

Creation procedures can be used as regular routines as well. Therefore, AutoProof will verify all creation routines twice, once as creation routines and once as regular routines. The context of the verification is different for the two verifications, as for example for creation routines all attributes are initialized to their default values before the routine is executed.

You can instruct AutoProof to verify a creation routine only once by adding a *creator* annotation. This denotes the routine as being creation-only and AutoProof will not verify it as a regular routine.

```
make
  note
    status: creator
  do ··· end
```

Marks `make` to be only a creation routine.

Functional Functions

AutoProof supports a special type of function, consisting of only a single assignment to **Result**. To declare such a function you have to add a *functional* annotation to the function. These functions are defined by their implementation and have an implicit postcondition; given an implementation **Result** := x the implicit postcondition will be **Result** = x.

```
available_amount: INTEGER
  note
    status: functional
  do
    Result := balance + credit_limit
  end
```

Marks `available_amount` to be *functional*, therefore only consisting of a single assignment to `Result`.

1.6 Debugging Verification

The only feedback given by AutoProof is whether a routine is successfully verified or if some specific assertions could not be proven. When the verification fails it can be necessary to find out which facts the verifier could establish or even guide the verifier to the right conclusion. For this you can use intermediate assertions (`check` instructions in Eiffel). During the debugging process it can also be beneficial to *assume* specific facts and thus limit the possible executions that the verifier considers during the proof.

Assertions

Using Eiffel's `check` instruction you can add an intermediate assertion that will be verified by AutoProof. This can help to check if you have the same understanding of the state at a program point as the verifier. You can add multiple expressions to a single check instruction, and each expression can be equipped with a tag. AutoProof will show the tags in error messages.

`check tag: expr end`	Check instruction to establish if `expr` holds.

Note that it is possible that when you have multiple consecutive assertions successfully verified, removing an intermediate assertion will make the verification of later assertions fail. In these cases you have to keep the assertion in order to guide the verifier towards the successful verification.

Assumptions

Eiffel does not support assumptions out of the box. To write an assumption in AutoProof, you have to write a check instruction with the special tag *`assume`*. AutoProof will assume the expression for the rest of the routine without checking it.

You can use assumptions to limit the executions considered by the verifier. For example by assuming `False` in a branch of a conditional instruction the verification of that branch will always succeed.

```
if ··· then
  ···
else
  check assume: False end
end
```

Ignores all code path that go through the else branch.

Another way to use assumptions to limit executions it by restricting the state space of otherwise unrestricted values. This can be used for example to ignore executions where an array is empty.

```
check assume: not a.is_empty end
```

Ignores executions where a is empty.

Inconsistencies

It can happen that verification succeeds due to inconsistent contracts or assumptions. If you for example have a routine with the precondition a >0 and an additional class invariant a <0 (or an assumption a <0 in the body of the routine), your specification is inconsistent. This is essentially equivalent to an assumption of False and the verifier will be able to derive any fact from it, including false ones.

A quick (though not completely safe) check for inconsistencies is to add an assertion or postcondition False to your routine. If the verifier manages to prove the assertion, this is a sign for an inconsistency in the specification.

1.7 Hands-On: Clock

The CLOCK class is modeling a clock counting seconds, minutes and hours of a day. The class contains routines to create the clock, set the time, and increase the time.

Task 1: Add a *model* declaration to define the abstract model.

Task 2: Add a class **invariant** to restrict the attribute values.

Task 3: Add a precondition to the creation procedure make.
You should be able to verify make and test_make.

Task 4: Add the specification to the set_∗ procedures.
You should be able to verify the set_∗ and test_set procedures.

Task 5: Add the specification to the increase_∗ procedures.
You should be able to verify both classes completely.

2 Verification of Algorithmic Problems

An important aspect in the verification of programs is verifying algorithms. In this section we will focus on the verification of algorithmic problems on arrays, such as searching and sorting. The concepts needed to verify array algorithms are also necessary for other types of algorithms.

We use the algorithm of finding the maximum element of an integer array as an example. The code is shown in Figure 2. You can look through the example again and verify it. In the rest of this section we will explain in detail how one verifies such an algorithm.

```
class MAX_IN_ARRAY
feature -- Basic operations

  max_in_array (a: SIMPLE_ARRAY [INTEGER]): INTEGER
      -- Find the maximum element of 'a'.
    require
      array_not_empty: a.count >0
    local
      i: INTEGER
    do
      Result := a[1]
      from
        i := 2
      invariant
        i_in_bounds: 2 ≤ i and i ≤ a.sequence.count + 1
        max_so_far: across 1 |..| (i−1) as c all a.sequence[c.item] ≤ Result end
        in_array: across 1 |..| (i−1) as c some a.sequence[c.item] = Result end
      until
        i >a.count
      loop
        if a[i] >Result then
          Result := a[i]
        end
        i := i + 1
      variant
        a.count − i
      end
    ensure
      is_maximum: across 1 |..| a.count as c all a.sequence[c.item] ≤ Result end
      in_array: across a.sequence.domain as c some a.sequence[c.item] = Result end
    end

end
```

Figure 2: *Maximum in array* example.

2.1 Mathematical Model Library

To express complex mathematical properties, AutoProof supports the *Mathematical Model Library*. This library consists of classes modeling sets, bags (or multisets), sequences, maps, intervals, and relations. You can find an API description of these classes online[3].

MML classes do not have an implementation and should therefore only ever be used for specifications (using *ghost fields* and *ghost code* as discussed in Section 2.6). They have an efficient axiomatization in the back-end verifier, and are therefore well suited to be used with AutoProof.

MML Types

The most important MML types are:

- `MML_SET [G]`: A set contains distinct objects. Each element can only be contained once and the order is irrelevant.
- `MML_SEQUENCE [G]`: A sequence is an ordered list of elements. Indexing starts at 1.
- `MML_BAG [G]`: A bag (or multiset) is a set where each element can appear multiple times. The order of elements is irrelevant.

Shorthand Notations

Several shorthand notations exist to declare sets and sequences making the use of MML classes easier.

- Sets of type `MML_SET [ANY]` can be declared using the Eiffel manifest tuple notation: `s := [a, b]`.
- Sets of type `MML_SET [G]` can be declared using the Eiffel manifest array notation: `s := ≪a, b≫`.
- Sequences of type `MML_SEQUENCE [G]` can be declared using the Eiffel manifest array notation: `s := ≪a, b≫`.
- Use `{MML_SET [G]}.empty_set` to declare an empty set.
- Use `{MML_SEQUENCE [G]}.empty_sequence` to declare an empty sequence.

For the last two shorthands it is not possible to use the empty array notation ≪≫ due to the intricacies of Eiffel typing.

[3] http://se.inf.ethz.ch/research/autoproof/reference/

2.2 SIMPLE_ARRAY and SIMPLE_LIST

When you want to use arrays or lists in verification, you need classes that have a fully specified interface. The classes from EiffelBase do not offer this, therefore when verifying algorithms with AutoProof, you should use the two provided classes SIMPLE_ARRAY [G] and SIMPLE_LIST [G]. Both classes have a ghost model field sequence of type MML_SEQUENCE [G] and all features are specified in terms of the model. You can find an API description of these classes online[4].

To make it easier for AutoProof to deal with specifications involving these classes you should use the sequence model field when writing complex assertions involving the container contents. For example the loop invariant of the max_in_array function is written as:

```
2 ≤ i and i ≤ a.sequence.count + 1
across 1 |..| (i−1) as c all a.sequence[c.item] ≤ Result end
across 1 |..| (i−1) as c some a.sequence[c.item] = Result end
```

Were you to replace a.sequence with just a, AutoProof would not verify the routine anymore (*try it!*).

2.3 Quantifiers

Eiffel supports bounded universal and existential quantifiers with the **across** expression. In our example, where we find the maximum in an array, we can use this to express the desired postcondition that all elements in the array are smaller or equal to the result. Universal quantification is done using the **across**..**all** expression. With the Eiffel interval expression 1 |..| a.count we can quantify over all integers between (and including) 1 and a.count.

```
across 1 |..| a.count as c all a.sequence[c.item] ≤ Result end
```

The across loop uses a cursor, therefore we have to use c.item to access the current element of the iteration. For the correctness of the algorithm we also have to express that the result is an element of the array, not just larger than all elements. We can do this with an existential quantification using Eiffel's **across**..**some** loop.

```
across a.sequence.domain as c some a.sequence[c.item] = Result end
```

In the last example we used the domain query for the quantification to show that you can use different approaches to reach the same goals. This query defined in MML_SEQUENCE returns a set of integer values that contains

[4] http://se.inf.ethz.ch/research/autoproof/reference/

all index values of the sequence and is therefore equivalent to using an interval from 1 to `a.count` (all MML sequences are indexed from 1).

AutoProof supports the following domains for quantification:

- Integer intervals. The quantified variable will be of type `INTEGER`.
- Sets of type `MML_SET [G]`. The quantified variable will be of type `G`.
- Sequences of type `MML_SEQUENCE [G]`. The quantified variable will be of type `G`. This is equivalent to quantifying over the `range` of a sequence.
- Objects of type `SIMPLE_ARRAY [G]` or `SIMPLE_LIST [G]`. The quantified variable will be of type `G`. This is equivalent to quantifying over the `sequence` of the array or list.

2.4 Termination

AutoProof will verify termination of loops and direct recursive calls (indirect recursion is not checked). To prove termination you can define *loop variants* for loops or *decreases clauses* for loops and recursive routines.

Loop Variant

The loop variant is an integer expression that is non-negative and decreases with each loop iteration. This implies that the loop can only be executed a finite number of times.

```
loop ···
variant
  a.count − i
end
```

The loop variant decreases each loop iteration and stays non-negative.

AutoProof infers loop variants of simple loops. For example a loop with exit condition `a >b` will have an inferred loop variant of `b − a`. In the example in Figure 2 specifying the variant is not necessary.

Decreases Clause

In complex algorithms it is possible that an integer value is not enough to express the loop variant. For these cases AutoProof supports *`decreases`* clauses. A decreases clause can contain multiple arguments of type `INTEGER`, `MML_SET`, or `MML_SEQUENCE`. The semantics of a decreases clause is that in each loop iteration the tuple that contains all the elements of the decreases clause needs to become lexicographically smaller while remaining bounded from below. The lower bound is 0 for integers and is the empty set or empty sequence for sets and sequences.

The decreases clause for a loop is written in the loop invariant.

```
from ···
invariant
  decreases (a.count − i)
until ···
```

Decreases clause equivalent to the previous loop variant.

For recursive functions the decreases clause is added to the precondition. Otherwise it behaves like the decreases clause for loops: at each recursive call the value of the decreases clause must become smaller while remaining bounded.

```
f (a: SET [INTEGER]; b: INTEGER)
  require
    decreases (a, b)
  do ··· end
```

Decreases clause of a recursive function.

Non-termination

Sometimes it is not desirable to prove termination of an algorithm. For these cases you can add an empty decreases clause to the loop or recursive function and AutoProof will skip the termination check.

```
from ···
invariant
  decreases ([])
until ···
```

A possibly non-terminating loop.

2.5 Hands-On: Linear and Binary Search

With the knowledge we have so far we now verify algorithms searching an element in an array. These algorithms do not change the array and are therefore *pure*, thus simplifying the specification.

Linear Search

Task 1: Add the loop variant to verify that the loop terminates.
You should be able to verify `linear_search` in its current form.

Task 2: Add postconditions to `linear_search` to verify the test class.
You should be able to verify the test class.

Task 3: Add loop invariants to verify the postcondition.
You should be able to verify both classes completely.

Binary Search

Task 1: Add loop invariants to verify that all array accesses are valid.

Task 2: Add the loop variant to verify that the loop terminates.
You should be able to verify `binary_search` in its current form.

Task 3: Add precondition to require input arrays to be sorted.

Task 4: Add postconditions to `binary_search` to verify the test class.
You should be able to verify the test class.

Task 5: Add loop invariants to verify the postcondition.
You should be able to verify both classes completely.

Recursive Binary Search

Task 1: Add the specification to `binary_search` (you can reuse the specification of the iterative version).
You should be able to verify the test class.

Task 2: Add precondition to `binary_search_recursive_step` to require the input array to be sorted and to verify that all array accesses are valid.

Task 3: Add a decreases clause to to prove termination of the recursion.
You should be able to verify `binary_search_recursive_step` in its current form.

Task 4: Add postconditions to `binary_search_recursive_step` to verify the algorithm.
You should be able to verify both classes completely.
Note: you might need intermediate assertions to verify the postcondition.

2.6 Ghost State and Ghost Functions

The next examples—iterative and recursive binary search—have preconditions that the input array is sorted. Writing an expression that expresses this property directly in the precondition can become unwieldy. It is beneficial to write helper functions that capture such properties with a meaningful name and that allow reuse of the function.

The *sorted* property was expressed over the *sequence* of the array, which is of type MML_SEQUENCE. As mentioned before (see Section 2.1), MML types are not executable and can only be used for specification purposes. We call code that is used only for specification purposes *ghost code*; ghost code is never executed and only interpreted by the verifier.

To write expressive specifications AutoProof supports ghost code in the form of *ghost functions*, using *ghost attributes*, and writing *lemmas*. Ghost code should never influence executable code, therefore assignments from ghost code to regular attributes is not allowed. AutoProof does **not** enforce this currently, so using ghost code outside specifications may lead to undefined behavior.

Ghost Functions

Ghost functions are useful to write helper functions usable in specifications, for example to express that a sequence is sorted. To mark a function as *ghost* you add a **note** clause with *status*: *ghost*. If the function is also *functional*, the note clause can be shortened by combining the two *status* properties.

```
is_sorted (s: MML_SEQUENCE [INTEGER]): BOOLEAN
    -- Is 's' sorted?
  note
    status: functional, ghost
  do
    Result := across 1 |..| s.count as i all
                across 1 |..| s.count as j all
                  i.item ≤ j.item implies s[i.item] ≤ s[j.item] end end
  end
```

With ghost function like the one shown above we can simplify contracts and promote reuse of specification constructs, for example in iterative and recursive binary search (*try it!*).

Ghost State

Ghost state is introduced by having *ghost attributes*. These attributes can be used like regular attributes in contracts, frame conditions, code, and as model fields. Most commonly you would use ghost attributes to define model fields that are then related to existing attributes or other objects through class invariants and other contracts. A linked list could for example represents its contents in form of a sequence using a ghost attribute to store the sequence and then declaring this attribute to be a model field.

To declare a ghost attribute you need to add a **note** clause to the attribute. The Eiffel syntax for doing this is the following:

```
sequence: MML_SEQUENCE [INTEGER]
  note status: ghost
  attribute
  end
```

Lemmas

Intermediate assertions are not always sufficient for difficult proofs. In these cases you can use *lemma* procedures to support verification. Calling a lemma procedure has the same effect as calling other procedures: the verifier asserts the precondition and assumes the postcondition. Lemmas can therefore be used to add $A(x) \implies B(x)$ to the fact space, where $A(x)$ is the precondition and $B(x)$ is the postcondition of the lemma.

Lemmas are implicitly *ghost* and *pure*. You declare a lemma using a special **note** clause.

```
lemma (x)
  note status: lemma
  require
    A(x)
  do
    -- Proof that A(x) implies B(x)
  ensure
    B(x)
  end
```

Lemmas are proven like a regular procedures. You might need to *implement* a proof; sometimes you can use recursion in a lemma which is akin to an induction proof.

2.7 Accessing Pre-state

Eiffel allows the use of **old** expressions in postcondition to express the effect of a routine in relation to the pre-state. This syntax is limited, as it cannot be used in **across** expressions or in the body of the routine (e.g. in loop invariants).

AutoProof offers an extension to the **old** mechanism through a ghost query *old_* defined in ANY. This query can be used anywhere in the code or in the postcondition to reference the value of an expression in the pre-state of the routine.

`check s.old_[i] = s[i] end`	Assertion that item `s [i]` is unchanged.

2.8 Integer Overflows

AutoProof can check a program for integer overflows. By default overflow checking is disabled. You can enable it among AutoProof's options or, if you use the command line version, with the `−overflow` command line option.

2.9 Hands-On: Sorting

The next examples are about sorting of arrays. The algorithms shown here are in-place algorithms that operate on the array directly. As a preliminary exercise we look at the notion of permutation of arrays and how to express this in AutoProof.

Permutation

Task 1: Find the correct encoding of permutation (only one is correct).

Task 2: For each incorrect encoding try to find two sequences that successfully pass the check instruction while not being real permutations.

Gnome Sort

Task 1: Add the frame specification, pre- and postcondition to `gnome_sort`.
Add implementation of `is_part_sorted`.
You should be able to verify the test class.

Task 2: Add loop invariants to verify that all array accesses are valid.

Task 3: Add loop invariants to verify the postcondition.
You should be able to verify both classes completely.

Task 4: Enable overflow checking and verity absence of overflows.

Insertion Sort

Task 1: Add a precondition and loop invariant to `insertion_sort` and the precondition to `swap` to verify that all array accesses are valid.

Task 2: Add the loop variants to verify that the loops terminate.
You should be able to verify `insertion_sort` and `swap` in its current form.

Task 3: Add the postcondition to `insertion_sort`.
You should be able to verify the test class.
You might want to introduce helper functions.

Task 4: Add the postcondition to `swap` and the necessary loop invariants to verify the postcondition.
You should be able to verify both classes completely.

Task 5: Enable overflow checking and verity absence of overflows.

3 Object Consistency and Ownership

For this section we use an unbalanced binary tree as an example. Each tree node has a value and one or two children. The `maximum` function returns the maximum element in the tree. An excerpt of the tree node class is shown in Figure 3.

3.1 State of an Object

AutoProof uses an invariant model where objects can be in a consistent state or in a potentially inconsistent state. Consistent objects are *closed* and their class invariants are guaranteed to hold. Inconsistent objects are *open* and their class invariant is potentially violated. Objects can only be modified when they are open, changing the value of an attribute is not allowed when an object is closed.

AutoProof supports a dynamic ownership model where objects can be *owned* by other objects. The ownership relations can evolve during runtime. Objects that are unowned or whose owner is open are called *free*. We define a shorthand for objects that are *closed* and *free* calling them *wrapped*. When an object is in a consistent state, the ownership tree rooted in that object is guaranteed to be consistent as well.

To model the object consistency and ownership relation, each objects has a boolean ghost field `closed`, a ghost field `owner` pointing to the potential owner of the object, and a ghost field `owns` that contains the set of all owned objects. The relationship between the `owns` set and `owner` field is guaranteed to hold for objects in a consistent state. A special case is **`Void`** which is always `open`. **`Void`** can therefore never be owned and must not be part of the `owns` set.

We illustrate the possible object states on the example object structure of Figure 4. The object structure consist of six objects:

- Object `a` is *closed*, therefore its class invariant is guaranteed to hold. It does not have an owner and thus is *free*. As it is both free and closed, it is also *wrapped*. The object `a` owns the two objects `b` and `c`. This is defined through its `owns` set, i.e. `a.owns` $=\{$`b`, `c`$\}$.

- Objects `b` and `c` are both owned by `a`, so their `owner` ghost field points to `a`. Since they are *owned* they are *not free*. As their owner `a` is *closed*, the two objects `b` and `c` are *closed* as well, as they are in `a`'s ownership domain.

```
class TREE_NODE

create
  make, make_with_children

feature {NONE} -- Initialization

  make (a_value: INTEGER)
      -- Initialize node.
    note
      status: creator
    do
      value := a_value
    ensure
      value_set: value = a_value
      no_left: left = Void
      no_right: right = Void
    end

  make_with_children (a_value: INTEGER;
          a_left, a_right: TREE_NODE)
      -- Initialize node.
    note
      status: creator
      explicit: contracts
    require
      a_left.is_wrapped
      a_right.is_wrapped

      modify (Current)
      modify_field ("owner", [a_left, a_right])
    do
      value := a_value
      left := a_left
      right := a_right
    ensure
      value_set: value = a_value
      left_set: left = a_left
      right_set: right = a_right
      default_is_closed: is_wrapped
    end

feature -- Access

  value: INTEGER
      -- Value of this node.

  left, right: TREE_NODE
      -- Left and right node (Void if none).

feature -- Basic operations

  maximum: INTEGER
      -- Maximum value of this tree.
    require
      decreases (sequence)
    do
      Result := value
      if left ≠ Void then
        check owns.has (left) end
        Result := Result.max (left.maximum)
      end
      if right ≠ Void then
        check owns.has (right) end
        Result := Result.max (right.maximum)
      end
    ensure
      max: across sequence.domain as i all
                sequence[i.item] ≤ Result end
      exists: sequence.has (Result)
    end

feature -- Model

  sequence: MML_SEQUENCE [INTEGER]
      -- Sequence of values.
    note
      status: ghost
    attribute
    end

invariant
  owns_def: owns = {like owns}[[left, right]] / Void
  sequence_def: sequence =
    (if left = Void
       then {like sequence}.empty_sequence
       else left.sequence end) +
    {like sequence}[≪value≫] +
    (if right = Void
       then {like sequence}.empty_sequence
       else right.sequence end)
end
```

Figure 3: Excerpt of the binary tree example.

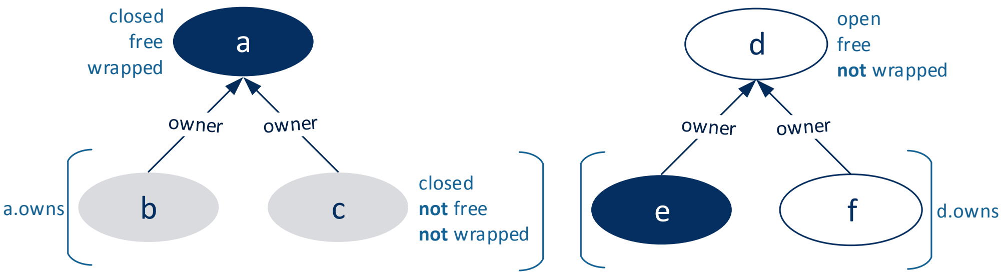

Figure 4: Example of an object structure at run-time.

- Object `d` is *open* and may potentially be in an inconsistent state, so its class invariant is not guaranteed to hold. It does not have an owner and is therefore *free*. The object `d` owns the two objects `e` and `f`, defined through its `owns` set.
- Object `e` is *closed* and therefore consistent. Its owner is `d`, but since `d` is *open*, `e` is considered to be *free*. Being both *closed* and *free* means that `e` is *wrapped*.
- Object `f` is *open* and potentially inconsistent. Analogous to `e` it is *free*, as its owner `d` is *open*.

The example illustrates the difference between ownership trees of *open* and *closed* objects. While ownership trees of consistent objects are guaranteed to be consistent as well—all objects in the ownership tree including the root object are *closed*—, this property does not hold for ownership trees of potentially inconsistent objects. The ownership tree of an *open* object may contain objects that are *open* and objects that are *closed*.

3.2 Object State Queries

AutoProof offers ghost functions that can be used to query an object's state in assertions and specifications:

- `is_wrapped: BOOLEAN` – is the object wrapped (closed and free)?
- `is_free: BOOLEAN` – is the object free (unowned or owner is open)?
- `is_open: BOOLEAN` – is the object open (potentially inconsistent)?
- `closed: BOOLEAN` – is the object closed (consistent)?

- owner: ANY – owner of the object.
- owns: MML_SET [ANY] – set of owned objects.
- inv: BOOLEAN – does full invariant of the object hold?
- inv_only (t): BOOLEAN – does the invariant with tag *t* hold?
- inv_without (t): BOOLEAN – does the invariant except for tag *t* hold?

The last two functions inv_only and inv_without allow to reuse specification constructs. The argument to these functions is a list of manifest strings containing invariant tags. Given the class of Figure 3 the condition that the sequence is consistent can be written as inv_only ("sequence_def"). This helps reduce the annotation burden for classes with comples class invariants.

3.3 Encoding Ownership

Ownership in AutoProof is used by adding objects to and removing objects from the owns set. The usual way of doing this is by defining the owns set as part of the class invariant. The binary tree example has the following class invariant:

```
owns_def: owns = {like owns}[[left, right]] / Void
```

The owns set consists of the two objects left and right unless they are **Void**. As in other situations, the encoding of the owns set influences the ability of AutoProof to reason about it. In the maximum function of the binary tree we have introduced two assertions owns.has (left) and owns.has (right) in the respective branches. Where we to remove these check instructions AutoProof would fail in verifying the function (*try it!*).

This is due to the use of the set removal operation / **Void**, which makes the reasoning about the set more difficult and forces us to help the verifier in the proof. Were we to use a different encoding of the owns set in the class invariant we could remove these assertions. The following encoding is more verbose but better suited for the verifier:

```
owns =
  if left = Void then
    if right = Void then {like owns}[[]] else {like owns}[[right]] end
  else
    if right = Void then {like owns}[[left]] else {like owns}[[left, right]] end
  end
```

With this encoding there is no need for the intermediate check instructions anymore (*try it!*).

3.4 Wrapping and Unwrapping

The two ghost procedures `wrap` and `unwrap` are used to change an object from being *unwrapped* to being *wrapped* and vice versa. Figure 5 gives an overview of how the object consistency changes when these procedures are called.

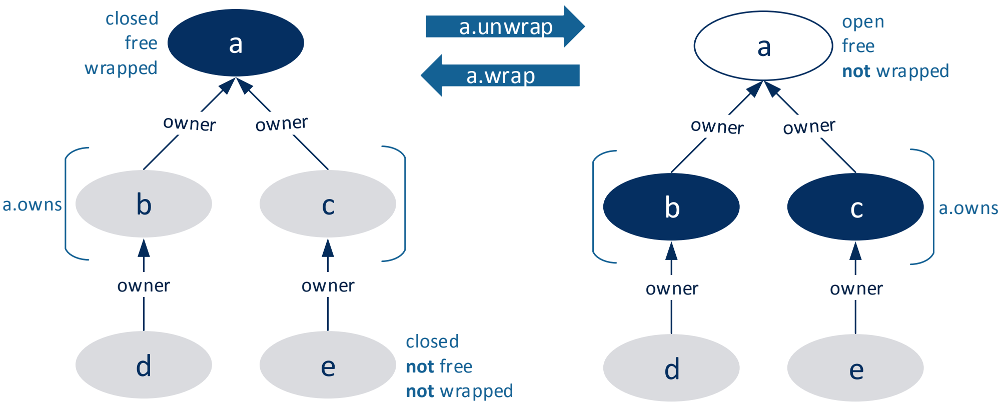

Figure 5: Change of object state on wrapping and unwrapping.

Since wrapping and unwrapping changes the boolean ghost field `closed`, that field must be writable when either of these procedures are called. This is also the reason we had to add the field `closed` to the modifies clause in the `withdraw` procedure of the *account* example (see Section 1.4).

`wrap` and `unwrap` are axiomatized in the background theory of the verifier. Their definition in the class `ANY` does therefore not reflect their real semantics. The actual specification for `wrap` and `unwrap` could be written as follows:

```
wrap
    -- Wrap 'Current'.
  require
    is_open
    inv
    across owns as o all o.item.is_wrapped end
    modify_field ("closed", Current)
    modify_field ("owner", owns)
  ensure
    is_wrapped
    across owns as o all o.item.owner = Current end
  end
```

```
unwrap
    -- Unwrap 'Current.
  require
    is_wrapped
    modify_field ("closed", Current)
  ensure
    is_open
    across owns as o all
          o.item.is_wrapped end
  end
```

3.5 Defaults

AutoProof uses implicit default contracts, default wrapping, and default assignments of ghost fields to remove the annotation burden. The default contracts and wrapping depend on type and visibility of routines. The default assignment to ghost fields depends on their definition.

Creation routines require by default that all arguments are wrapped. In addition, the `Current` object will be implicitly wrapped at the end of the routine body and the postcondition will assert that `Current` is wrapped.

```
make ($args$)
  require
    $\forall o \in args : o$.is_wrapped
  do
    ...
    Current.wrap
  ensure
    Current.is_wrapped
    $\forall o \in args : o$.is_wrapped
  end
```

Pure functions can by default be called on *closed* objects. It is therefore not necessary to unwrap the object before calling a function. Also there is no default postcondition since the objects are not mutated during the execution.

```
pure_function ($args$): $type$
  require
    Current.is_closed
    $\forall o \in args : o$.is_closed
  do ··· end
```

Public procedures and public impure functions require by default that all objects are in a consistent state, as they are callable from client code.

```
public_procedure ($args$)
  require
    Current.is_wrapped
    $\forall o \in args : o$.is_wrapped
  do
    Current.unwrap
    ...
    Current.wrap
  ensure
    Current.is_wrapped
    $\forall o \in args : o$.is_wrapped
  end
```

Private procedures and private impure functions are only callable from a restricted set of clients, for example from the class itself. Since all public routines by default unwrap the **`Current`** object, private routines by default assume the object to be open.

```
private_procedure (args)
  require
    Current.is_open
  do ···
  ensure
    Current.is_open
  end
```

Ghost fields that have a definition in the class invariant of the form *fieldname*=*expression* will be assigned implicitly every time the object is wrapped. This is the reason that the two ghost fields `owns` and `sequence` of the binary tree example are never explicitly assigned. AutoProof assigns these fields implicitly at the end of each procedure when the implicit call to `wrap` takes place.

Functional functions are often used in contracts and class invariants. Therefore they may operate on inconsistent objects and have no default preconditions.

Disabling Defaults

It is possible to disable defaults by adding the following **`note`** clauses to routines and classes:

- *`explicit`*: *`contracts`* – disable default pre- and postcondition; applicable to a routine or a class.
- *`explicit`*: *`wrapping`* – disable default `unwrap` and `wrap` instructions; applicable to a routine or a class.
- *`explicit`*: *fieldname* – disable default assignment to field *fieldname* before wrapping; applicable to a class.

3.6 Modification of Owned Objects

Modifies clauses define a set of modifiable objects. Whenever an object is modifiable then the *transitive closure* of all owned objects is also modifiable. It is therefore not necessary to add owned objects to a modifies clause when the owner is modifiable. A modifies clause of ***modify*** (o) is sufficient to potentially modify objects owned by o. Modifications of owned objects still require that the modified object is unwrapped.

3.7 Hands-On: Ring Buffer

The next exercise is about implementing a ring buffer backed by an array. This will highlight how to use ownership and model-based contracts to design a data structure.

Task 1: Add ownership definition for the data array to the class invariant.

Task 2: Add class invariants for the bounds of start and free.

Task 3: Add the model fields, model declaration, and class invariant describing the model.

Task 4: Add the specifications to the routines of RING_BUFFER. The tests in the test class can help you to get the specification right.
You should be able to verify both classes completely.