EECS-3311 – Lab2 – Abstract Graph

1/25/20 7:58:37 PM

1 Goals

```
require
     Lab0 done
     Lab1 done
     read accompanying document: Eiffel-101
ensure
     submitted on time
     no submission errors
rescue
     ask for help during scheduled labs
     attend office hours for TA Connor & Kevin
```

In the first month of the course, you are required to submit a sequence of three labs:

- Lab1: *List Graph*. Based on material on graph structures and algorithms from EECS2030, and EECS2011. Also introduces you to GRAPH [G] of Mathmodels where G is a generic parameter and GRAPH is a mathematical model of graphs.
- Lab2: *Abstract Graph*. Expands functionality of the LIST_GRAPH class to include more advanced queries.
- Lab3: *Maze*. Combines material from Lab1 and Lab2 into an interactive game using graph structures.

This Lab and others are accompanied by the accompanying document *Eiffel-101* which you must read and understand in the first 3 weeks of the term.

Goals of Lab2:

Primary

- See how Mathmodels is used to create mathematically complete contracts by using an **abstraction function**.
- Implement a topological sorting algorithm for the LIST_GRAPH class.

Secondary

- Further develop familiarity and competence with Eiffel: the language, method and tools
- Review and code with basic OO concepts such as polymorphism, dynamic binding, static typing, generic parameters and recursion
- Use *Design by Contract* (preconditions, postconditions and class invariants) for specifying API's and Unit Testing (test driven development and regression testing) for developing reliable code.
- Constructing implementations that satisfy *specifications*
- Tools Use: Use the IDE to browse code, edit, compile, unit test with ESpec, document the design, and develop competence in the use of the debugger
- Understand BON class diagrams for describing design decisions

- Use genericity and void safe programming constructs (**attached** and **detachable**) for abstraction and reliability
- Introduce a few design pattern examples (iterator and template)

2 Getting started

These instructions are for when you work on one of the EECS Linux Workstations or Servers (e.g *red*). You should not compile on *red* as it is a shared server; compile on your workstation. Do this by creating a folder in the tmp directory ($ > mkdir /tmp/eecs_username), and compiling to that directory. Invoke the Eiffel IDE from the command line as `estudio19.05` (aliased to `estudio`) and the command line compiler is `ec19.05` (aliased to `ec`).

2.1 Retrieve and compile Lab2

```
> ~sel/retrieve/3311/lab2
```

This will provide you with a starter directory `abstract-graph`. The directory has the following structure:

Table 1 Lab2 Directory Structure

```
abstract-graph/
├─── model
|    └─ list_graph.e
|    └─ vertex.e
|    └─ edge.e
|    └─ edge_comparator.e
├─── root
|    └─ root.e
├─── abstract-graph.ecf
└─ tests
      ├─── instructor
      |    └─ test_vertex_instructor.e
      |    └─ test_list_graph_instructor.e
      |    └─ course.e
      └─ student
           └─ student_test.e
```

The classes in **`red`** are **incomplete**, and you are required to complete all the *features* (queries and commands) in this classes, given the *specifications* (preconditions, postconditions and class invariants). Do not change the other classes.

You can now compile the Lab by running the following command and opening EStudio.

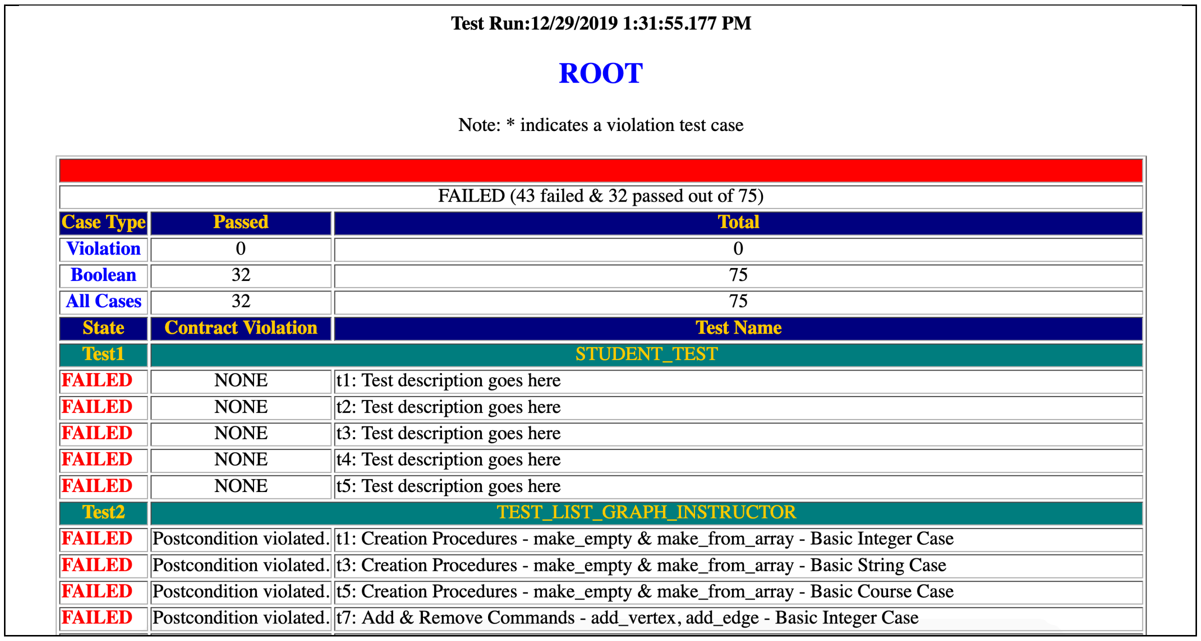
```
> estudio abstract-graph/abstract-graph.ecf &
```

where `abstract-graph.ecf` is the Eiffel configuration file for this Lab. The root class (file `root.e`) looks like this:

```
class
        ROOT

inherit
        ARGUMENTS_32

        ES_SUITE

create
        make

feature {NONE} -- Initialization

        make
                        -- Run app
                do
                        add_test (create {STUDENT_TEST}.make)
                        add_test (create {TEST_LIST_GRAPH_INSTRUCTOR}.make)
                        add_test (create {TEST_VERTEX_INSTRUCTOR}.make)
                        show_browser
                        run_espec
                end

end
```

Figure 1: Root class of abstract-graph

The project will compile and when you execute the Lab in workbench mode (Control-Alt-F5)[1], you will see the following *ESpec* Unit Testing report:



Test Run:12/29/2019 1:31:55.177 PM

ROOT

Note: * indicates a violation test case

🟥	🟥	🟥
FAILED (43 failed & 32 passed out of 75)		
Case Type	**Passed**	**Total**
Violation	0	0
Boolean	32	75
All Cases	32	75
State	**Contract Violation**	**Test Name**
Test1	STUDENT_TEST	
FAILED	NONE	t1: Test description goes here
FAILED	NONE	t2: Test description goes here
FAILED	NONE	t3: Test description goes here
FAILED	NONE	t4: Test description goes here
FAILED	NONE	t5: Test description goes here
Test2	TEST_LIST_GRAPH_INSTRUCTOR	
FAILED	Postcondition violated.	t1: Creation Procedures - make_empty & make_from_array - Basic Integer Case
FAILED	Postcondition violated.	t3: Creation Procedures - make_empty & make_from_array - Basic String Case
FAILED	Postcondition violated.	t5: Creation Procedures - make_empty & make_from_array - Basic Course Case
FAILED	Postcondition violated.	t7: Add & Remove Commands - add_vertex, add_edge - Basic Integer Case

Figure 2: ESpec Unit testing report from inital compilation (there are more tests than shown in this image)

[1] Read *Eiffel-101* for details

2.2 Get the initial Unit Tests working

The Red Bar means that some of the tests fail. You must get all the tests to work and obtain a Green Bar.

- STUDENT_TEST: you must write your own tests and we will check your tests. You may insert as many tests as you wish in this class.
- TEST_VERTEX_INSTRUCTOR: We provide you with some basic tests to ensure all of the features of VERTEX are working. You do not need to modify anything in VERTEX, since all of these tests should pass from the start.
- TEST_LIST_GRAPH_INSTRUCTOR: We provide you with tests for both basic commands and queries in LIST_GRAPH, as well as the advanced queries that were not included in Lab 1.

In order to get the initial tests to pass for TEST_LIST_GRAPH_INSTRUCTOR, you must first complete the basic features from Lab 1 and **implement the abstraction function.**

2.3 Implementing the abstraction function

In this Lab, one of the primary objectives is understanding how to use an abstraction function in order to be able to use Mathmodels contracts.

In Lab 1, it was mentioned that the contracts used were not mathematically complete, since it was not always possible to express the desired postconditions of features using "classical" contracts (using Eiffel alone), and oftentimes quite verbose.

In lab 2, we introduce the concept of Mathmodels contracts which address this issue. Mathmodels is a library that contains mathematical representations of data structures – for example, the mathematical model of a graph is simply a set of vertices and a relation of edges. This means we can perform mathematical operations (including forming predicates) on our code.

An example of this in practice can be seen by comparing the *add_vertex* contracts:

Classical:

```
add_vertex (a_vertex: VERTEX [G])
		require
			cl_non_existing_vertex: not has_vertex (a_vertex)
		do
			vertices.extend (a_vertex)
		ensure
			cl_add_vertex_membership: has_vertex (a_vertex)
			cl_add_vertex_others_unchanged: across
					(old vertices.deep_twin) is l_v
				all
					has_vertex (l_v)
				end
			cl_add_vertex_count: vertex_count = old vertex_count + 1
		end
```

Figure 3: add_vertex with classical contracts

Mathmodels:

```
add_vertex (a_vertex: VERTEX [G])
            require
                  mm_non_existing_vertex: not model.has_vertex (a_vertex)
            do
                  vertices.extend (a_vertex)
            ensure
                  mm_vertex_added: model ~ (old model.deep_twin) + a_vertex
            end
```

Figure 4: add_vertex with Mathmodels contracts

Note: that in these code snippets, Mathmodels contracts are prefixed by "mm", and classical contracts are prefixed by "cl".

As can be seen, the Mathmodels contract is able to capture the postcondition in a single statement, whereas the classical contract takes 3! More importantly, it describes a **complete** description of the postcondition, whereas the classical version is fragmented.

However, as you can see, the Mathmodels contract calls a `model` object. This is what is referred to as the **abstraction function** and must be completed before the LIST_GRAPH tests will pass.

```
feature -- Model

model: COMPARABLE_GRAPH [VERTEX [G]]
                  -- abstraction function
                  -- This must be implemented so that the contracts will work properly
                  -- You must find a way to translate the LIST_GRAPH implementation into the mathematical
                  -- model representation of a graph: COMPARABLE_GRAPH (which inherits from GRAPH).
            do
                  create Result.make_empty
            ensure
                        comment ("Establishes model consistency invariants")
            end
```

Figure 5: {LIST_GRAPH}model - the abstraction function for LIST_GRAPH

Why do we need the abstraction function?

The point of the abstraction function is to convert the LIST_GRAPH object to a COMPARABLE_GRAPH object that will be used in the contracts. Remember, we want our contracts to be mathematical statements to *ensure* that our features are correct. In order to write mathematical statements, we need to convert our implementation (the LIST_GRAPH adjacency list representation of a graph) into a mathematical representation of a graph.

Why not just use Mathmodels in the implementation?

We could use the Mathmodels classes as our implementation, however, since these classes include a lot of contract checking to act as complete specifications, they do not result in efficient implementations. Although they are not efficient in implementation, they are extremely useful for use in contracts to ensure the correctness of our implementation. To use Mathmodels contracts however, we have to create a way to convert our efficient implementation into the corresponding Mathmodels representation – which requires an abstraction function.

Your job is to complete this function so that the result of the model function will return a COMPARABLE_GRAPH object that has all the same vertices and edges of the current LIST_GRAPH object.

Note: Mathmodels classes contain both mutable commands (which you will use in your abstraction function), and immutable queries which allows the contracts to perform operations that can verify the correctness of your implementation without changing the state. For example, vertex_extend is a command to extend the GRAPH with a new vertex, but vertex_extended is a query that does not change the state of the GRAPH. Keep this in mind to avoid confusion when writing your abstraction function.

Please refer to Eiffel-101 and the accompanying *graph-algorithms.pdf* for further details on the abstraction function. The *graph-algorithms.pdf* contains an example of an abstraction function that converts the implementation of a stack to a Mathmodels SEQ (sequence), as shown in Fig. 6.

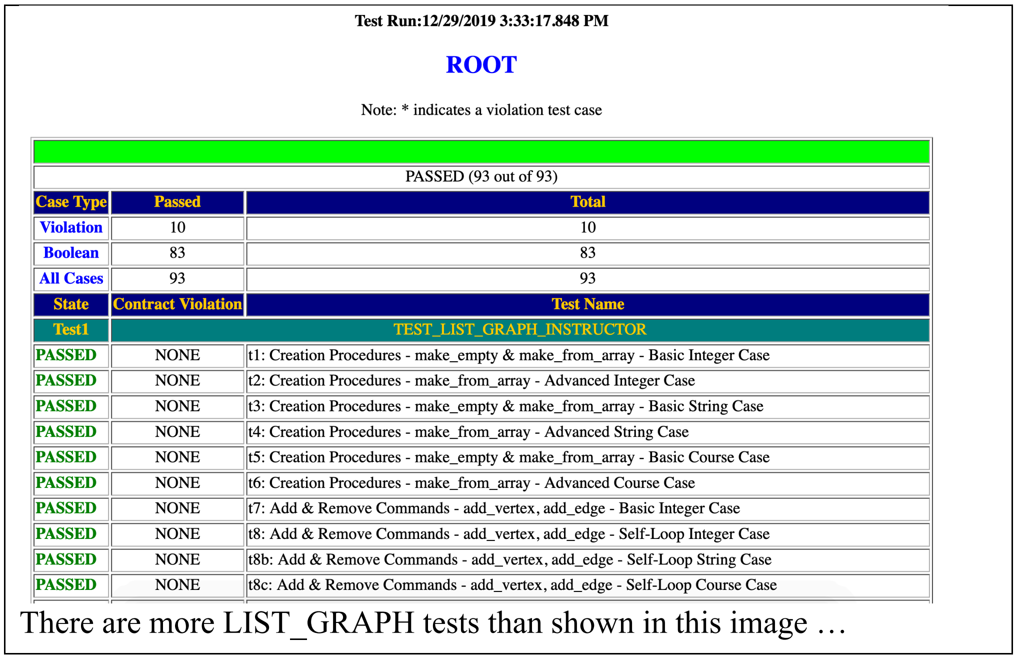
```
implementation: ARRAY [G]
model: SEQ [G] -- abstraction function
  do
    create Result.make_empty
    from i := implementation.lower
    until i > implementation.upper
    loop
      Result.prepend (implementation[i])
      i := i + 1
    end
  end
```

Figure 6: Example Abstraction function to convert a stack into a SEQ.

2.4 Get the remaining Unit Tests working

Once you have completed the abstraction function and the features from the previous Lab, you will be left with the *topologically_sorted* and *is_topologically_sorted* features. Complete these features to pass the remaining tests.

Important Note: *To check syntax, a compile (shortcut F7) is sufficient. But it is best to freeze (Control-F7) before running unit tests. Run the unit tests often! (even after very small changes to your code). When you compile, ensure that the compilation succeeded (reported at the bottom of the IDE). When you run unit tests, ensure that all your routines terminate (can also be checked in the IDE). If you keep running the tests without halting the current non-terminating run, you will keep adding new non-terminating processes to the workstation, and the workstation will choke on all the concurrently executing processes. Study how to use your tools effectively and efficiently.*



Test Run:12/29/2019 3:33:17.848 PM

ROOT

Note: * indicates a violation test case

🟩

PASSED (93 out of 93)

Case Type	Passed	Total
Violation	10	10
Boolean	83	83
All Cases	93	93
State	**Contract Violation**	**Test Name**
Test1		TEST_LIST_GRAPH_INSTRUCTOR
PASSED	NONE	t1: Creation Procedures - make_empty & make_from_array - Basic Integer Case
PASSED	NONE	t2: Creation Procedures - make_from_array - Advanced Integer Case
PASSED	NONE	t3: Creation Procedures - make_empty & make_from_array - Basic String Case
PASSED	NONE	t4: Creation Procedures - make_from_array - Advanced String Case
PASSED	NONE	t5: Creation Procedures - make_empty & make_from_array - Basic Course Case
PASSED	NONE	t6: Creation Procedures - make_from_array - Advanced Course Case
PASSED	NONE	t7: Add & Remove Commands - add_vertex, add_edge - Basic Integer Case
PASSED	NONE	t8: Add & Remove Commands - add_vertex, add_edge - Self-Loop Integer Case
PASSED	NONE	t8b: Add & Remove Commands - add_vertex, add_edge - Self-Loop String Case
PASSED	NONE	t8c: Add & Remove Commands - add_vertex, add_edge - Self-Loop Course Case

There are more LIST_GRAPH tests than shown in this image …

Figure 7: Green Bar showing all tests have passed.

2.5 List of features to implement to Specifications

Complete the following features of LIST_GRAPH:

The abstraction function:

- `model`

Features from Lab 1:

- `get_vertex`
- `vertex_count`
- `edge_count`
- `is_empty`
- `has_vertex`
- `has_edge`
- `edges`
- `reachable`
- `add_vertex`
- `add_edge`
- `remove_edge`
- `remove_vertex`

Advanced Features for Lab 2:

- `topologically_sorted`
- `is_topologically_sorted`

Notes:

- The features from Lab 1 should pass by simply using your previous implementation (assuming they are correct) and completing the abstraction function.
- If you are getting postcondition violations for features that passed the classical contracts, remember to re-check both the implementation of that feature as well as your abstraction function.
- There is an additional class called COURSE in the test folder. This class is used in some of the tests to ensure the programs work across custom classes as well as base classes such as INTEGER and STRING.

2.6 Documenting Architecture: BON/UML Class Diagrams

A critical way to document a design (and the design decisions) is via a BON class diagram. Use the EiffelStudio IDE to generate BON (or UML) class diagrams. For our Lab, the IDE generates the following:

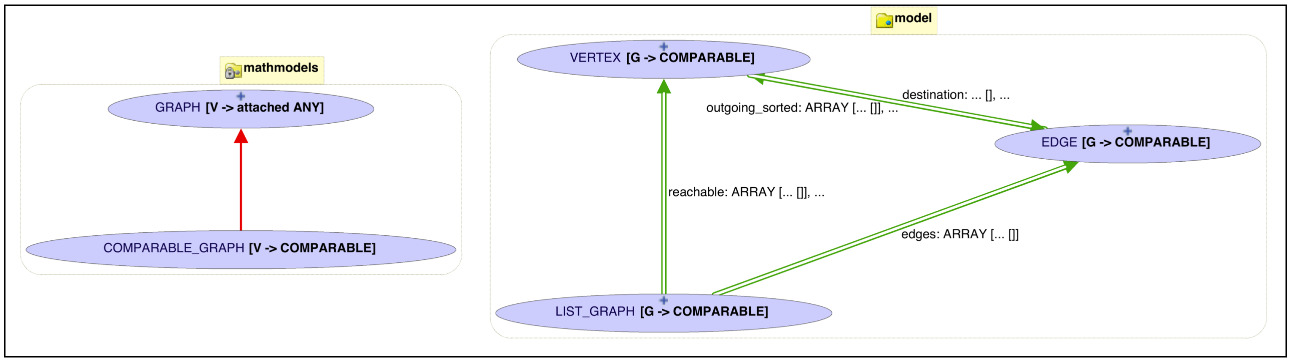

mathmodels
GRAPH [V -> attached ANY]
COMPARABLE_GRAPH [V -> COMPARABLE]
model
VERTEX [G -> COMPARABLE]
destination: ... [], ...
outgoing_sorted: ARRAY [... []], ...
EDGE [G -> COMPARABLE]
reachable: ARRAY [... []], ...
edges: ARRAY [... []]
LIST_GRAPH [G -> COMPARABLE]

Figure 8: BON class diagram (IDE generated)

This diagram shows some important characteristics of the design:

- The diagram shows one cluster: *model* (don't get this confused with the abstraction function model!). Each cluster contains classes (shown as ellipses). The green double arrows denote *client-supplier* relationships. A red single arrow denotes an *inheritance* relationship between classes, as seen from COMPARABLE_GRAPH to GRAPH.
- A "*" decorator denotes *deferred* classes and the "+" decorator denotes *effective* classes. A deferred class has at least one *routine* (either a query or a command) that is deferred, i.e. has no implementation. Such a class cannot be instantiated at runtime, and thus does not have explicit constructors.
- Classes are always written using UPPER_CASE. *Features* (queries and commands) are written using lower case.
- Class LIST_GRAPH [G] has one *generic* parameter, G for the type of element stored in the vertices. Generic parameter G is *constrained* to be COMPARABLE, needed for a sorted order.

As shown above, class LIST_GRAPH has an *edges* query containing all the edges in the graph, and a *reachable* query that returns all vertices that are reachable from the specified vertex. There is also a *vertices* query that contains all the vertices in the graph (not shown in the diagram).

The VERTEX class has an *outgoing_sorted* query that returns all of the outgoing edges from the specified vertex. The *source* and *destination* queries in EDGE are supplied by the VERTEX class.

Throughout the implementations you are required to code, there are often contracts that the implementation must satisfy. In this Lab, we have seen that with Mathmodels we can provide complete contracts and have omitted the classical contracts altogether.

2.7 Design by Contract (DbC), Class invariants and Iterator Design Pattern

Class VERTEX [G → COMPARABLE] provides contracts for commands such as *add_edge*.. This class also has *invariants* that must be satisfied by all routines (whether function routines or command routines). An example of an invariant for this class is:

$\forall e \in \texttt{outgoing}: \texttt{e.source} = \texttt{Current}$

The invariant asserts that each outgoing edge has Current as the source vertex. In Eiffel, the invariant is written as shown below in Figure 9.

```
invariant
      outgoing_edges_start_with_current:
            across
                  outgoing as l_edge
            all
                  l_edge.item.source ~ Current
            end
```

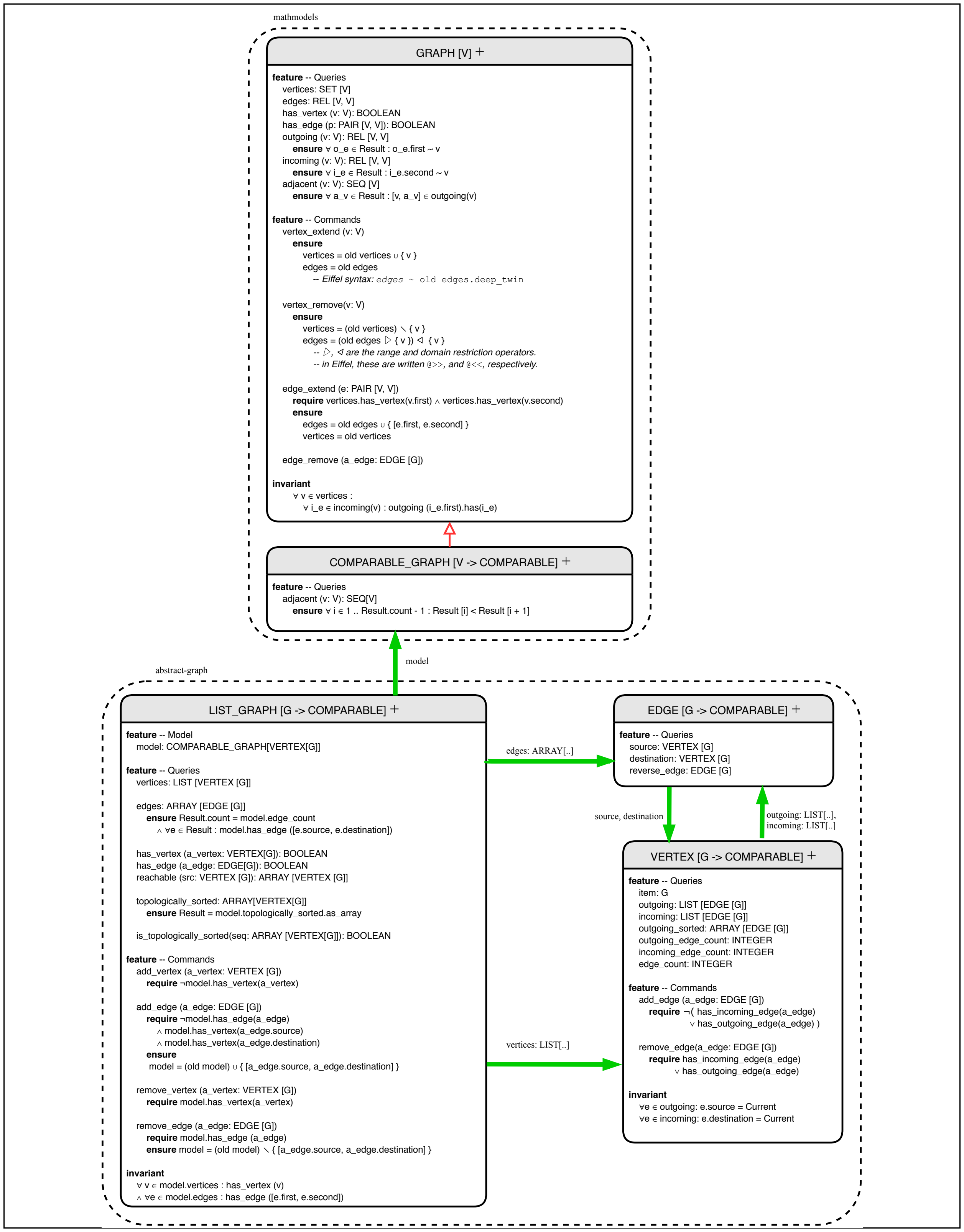
Figure 9: Class invariant for VERTEX, described in Eiffel across notation

The universal quantifier $\forall n$ is written using the **across** construct. This construct is based on the *iterator design pattern*.

2.8 Design architecture: BON diagram using the draw.io Template

The IDE Drawing tool is a good starting point for the BON class diagram. But we obtain a better view of the design using the draw.io tool.[2]

[2] http://seldoc.eecs.yorku.ca/doku.php/eiffel/faq/bon



```
GRAPH [V] +

feature -- Queries
   vertices: SET [V]
   edges: REL [V, V]
   has_vertex (v: V): BOOLEAN
   has_edge (p: PAIR [V, V]): BOOLEAN
   outgoing (v: V): REL [V, V]
      ensure ∀ o_e ∈ Result : o_e.first ~ v
   incoming (v: V): REL [V, V]
      ensure ∀ i_e ∈ Result : i_e.second ~ v
   adjacent (v: V): SEQ [V]
      ensure ∀ a_v ∈ Result : [v, a_v] ∈ outgoing(v)

feature -- Commands
   vertex_extend (v: V)
      ensure
         vertices = old vertices ∪ { v }
         edges = old edges
            -- Eiffel syntax: edges ~ old edges.deep_twin

   vertex_remove(v: V)
      ensure
         vertices = (old vertices) ∖ { v }
         edges = (old edges ▷ { v }) ◁ { v }
            -- ▷, ◁ are the range and domain restriction operators.
            -- in Eiffel, these are written @>>, and @<<, respectively.

   edge_extend (e: PAIR [V, V])
      require vertices.has_vertex(v.first) ∧ vertices.has_vertex(v.second)
      ensure
         edges = old edges ∪ { [e.first, e.second] }
         vertices = old vertices

   edge_remove (a_edge: EDGE [G])

invariant
      ∀ v ∈ vertices :
         ∀ i_e ∈ incoming(v) : outgoing (i_e.first).has(i_e)
```

```
COMPARABLE_GRAPH [V -> COMPARABLE] +

feature -- Queries
   adjacent (v: V): SEQ[V]
      ensure ∀ i ∈ 1 .. Result.count - 1 : Result [i] < Result [i + 1]
```

```
LIST_GRAPH [G -> COMPARABLE] +

feature -- Model
   model: COMPARABLE_GRAPH[VERTEX[G]]

feature -- Queries
   vertices: LIST [VERTEX [G]]

   edges: ARRAY [EDGE [G]]
      ensure Result.count = model.edge_count
         ∧ ∀e ∈ Result : model.has_edge ([e.source, e.destination])

   has_vertex (a_vertex: VERTEX[G]): BOOLEAN
   has_edge (a_edge: EDGE[G]): BOOLEAN
   reachable (src: VERTEX [G]): ARRAY [VERTEX [G]]

   topologically_sorted: ARRAY[VERTEX[G]]
      ensure Result = model.topologically_sorted.as_array

   is_topologically_sorted(seq: ARRAY [VERTEX[G]]): BOOLEAN

feature -- Commands
   add_vertex (a_vertex: VERTEX [G])
      require ¬model.has_vertex(a_vertex)

   add_edge (a_edge: EDGE [G])
      require ¬model.has_edge(a_edge)
         ∧ model.has_vertex(a_edge.source)
         ∧ model.has_vertex(a_edge.destination)
      ensure
       model = (old model) ∪ { [a_edge.source, a_edge.destination] }

   remove_vertex (a_vertex: VERTEX [G])
      require model.has_vertex(a_vertex)

   remove_edge (a_edge: EDGE [G])
      require model.has_edge (a_edge)
      ensure model = (old model) ∖ { [a_edge.source, a_edge.destination] }

invariant
   ∀ v ∈ model.vertices : has_vertex (v)
   ∧ ∀e ∈ model.edges : has_edge ([e.first, e.second])
```

```
EDGE [G -> COMPARABLE] +

feature -- Queries
   source: VERTEX [G]
   destination: VERTEX [G]
   reverse_edge: EDGE [G]
```

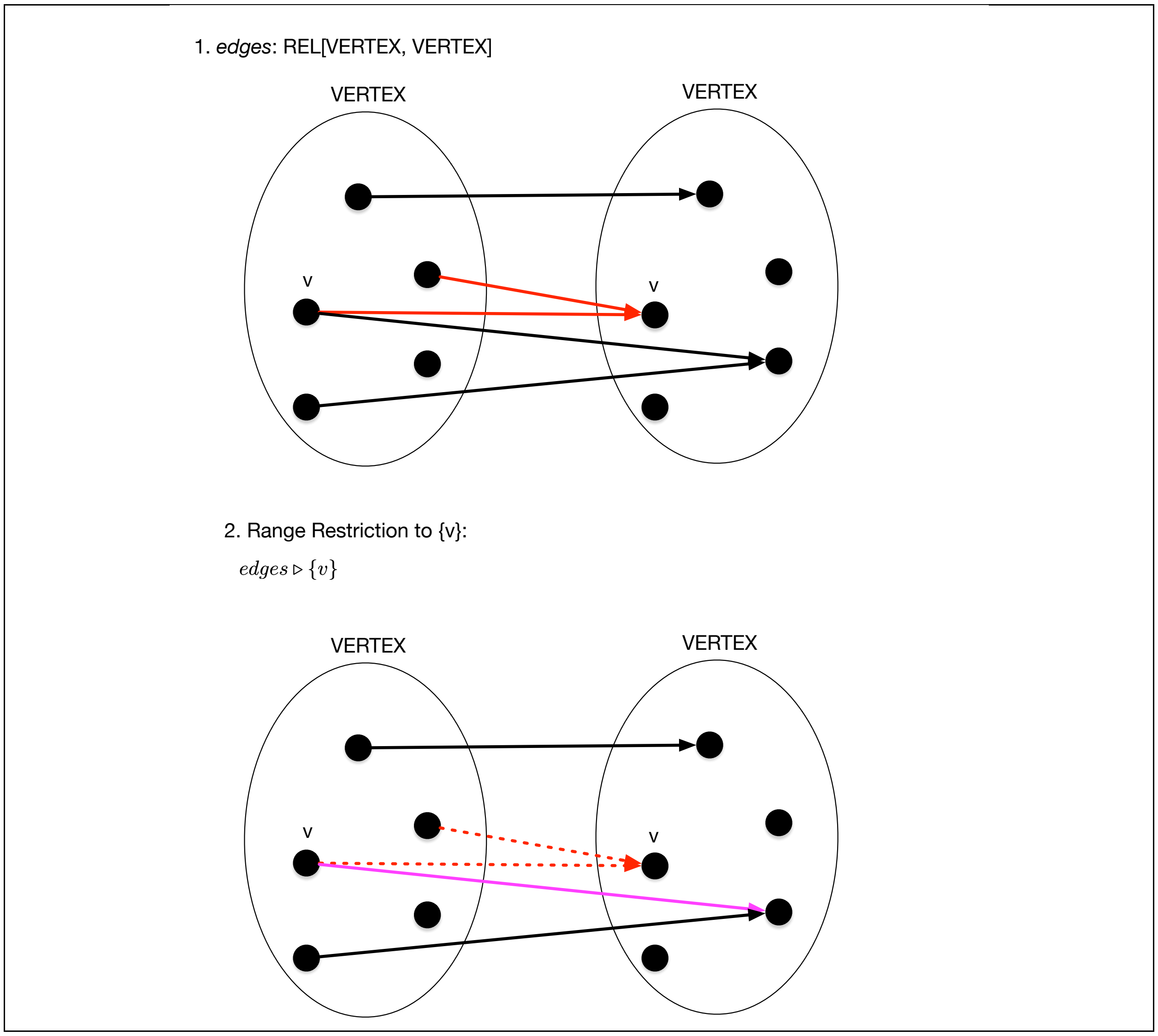
```
VERTEX [G -> COMPARABLE] +

feature -- Queries
   item: G
   outgoing: LIST [EDGE [G]]
   incoming: LIST [EDGE [G]]
   outgoing_sorted: ARRAY [EDGE [G]]
   outgoing_edge_count: INTEGER
   incoming_edge_count: INTEGER
   edge_count: INTEGER

feature -- Commands
   add_edge (a_edge: EDGE [G])
      require ¬( has_incoming_edge(a_edge)
               ∨ has_outgoing_edge(a_edge) )

   remove_edge(a_edge: EDGE [G])
      require has_incoming_edge(a_edge)
            ∨ has_outgoing_edge(a_edge)

invariant
   ∀e ∈ outgoing: e.source = Current
   ∀e ∈ incoming: e.destination = Current
```

Figure 10: BON class diagram (draw.io template)

The draw.io diagram is constructed manually, which allows for the selective provision of classes, their relationships, their features, their signatures, their contracts and class invariants. The design architecture is thereby better described.

2.9 Restriction Operators

In the BON diagram, you will see that the {GRAPH}.vertex_remove command uses the domain and range restriction operators. Fig. 11 contains an example of a range restriction being applied to a relation of edges, then a domain restriction.


1. *edges*: REL[VERTEX, VERTEX]
VERTEX
VERTEX
v
v
2. Range Restriction to {v}:
$edges \triangleright \{v\}$
VERTEX
VERTEX
v
v

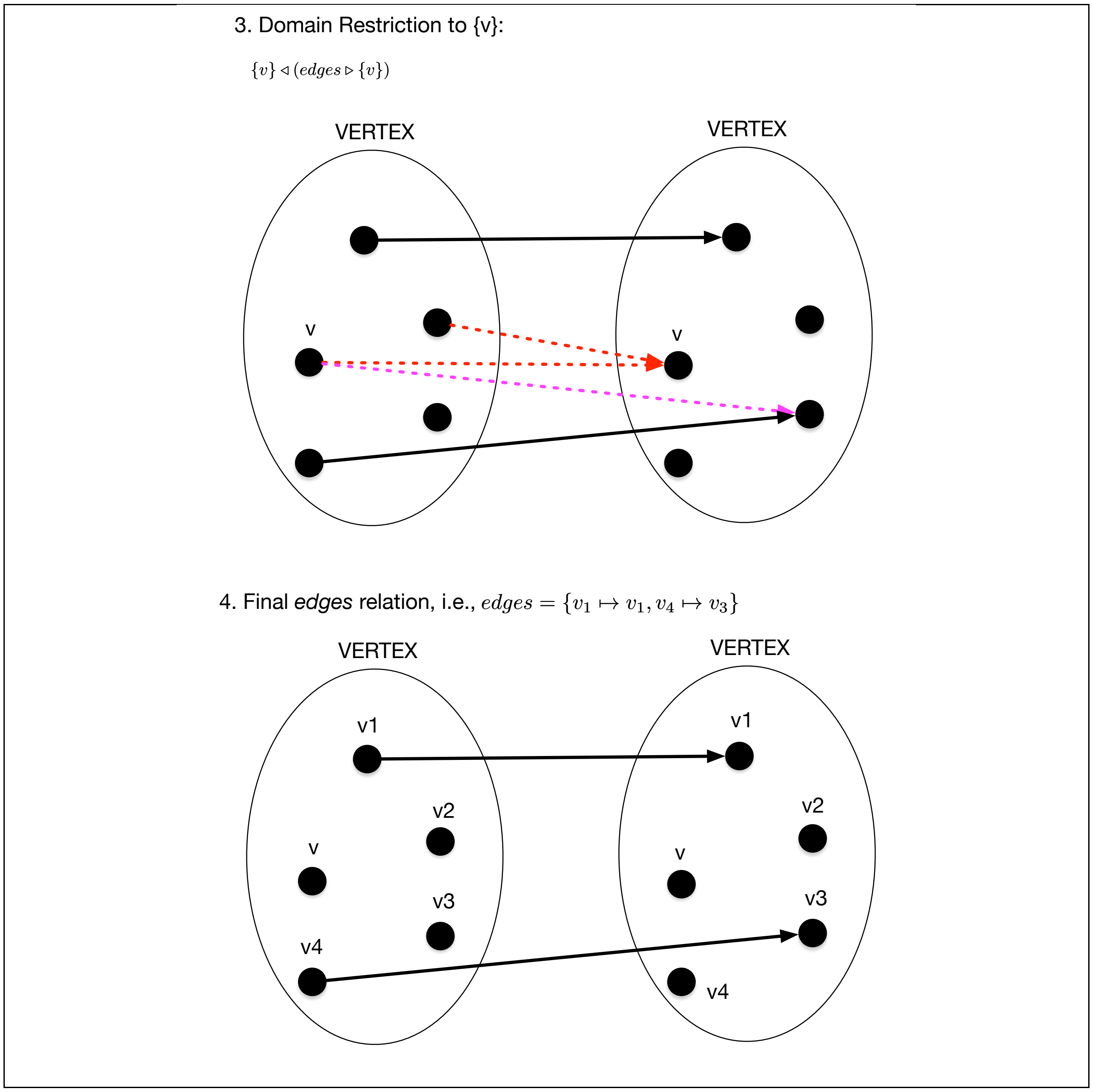

3. Domain Restriction to {v}:
$\{v\} \lhd (edges \rhd \{v\})$
VERTEX
VERTEX
v
v
4. Final *edges* relation, i.e., $edges = \{v_1 \mapsto v_1, v_4 \mapsto v_3\}$
VERTEX
VERTEX
v1
v2
v
v3
v4
v1
v2
v
v3
v4

Figure 11: Restriction Operand Example

As can be seen in Fig. 10, *edges* is a relation between two sets of vertices. First, a range restriction is applied, which means we create a new relation only containing edges that do not have *v* in the range.

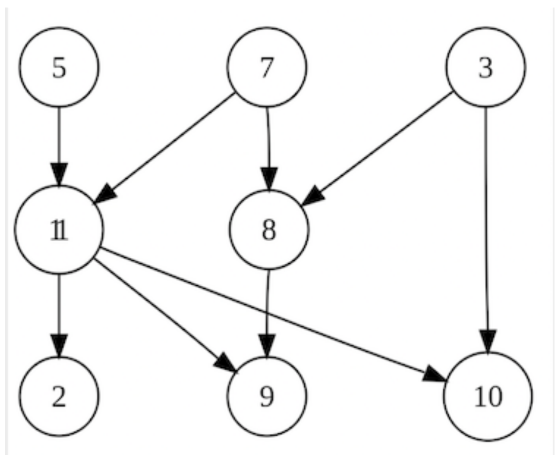
Then, a domain restriction is applied. This means that we now create a new relation that is the same as the previous one, but only including edges in it that do not have *v* in the domain.

In the end, we are left with a relation of edges that does not contain the element *v* at all.

3 Graphs - Topological Sort

Topological sorting has a variety of real-world applications, including job scheduling, shortest-path finding, and cycle-detection.

Expanding upon our work with graphs from last week, we extend the functionality of our LIST_GRAPH class by including a *topologically_sorted* feature. Any directed, acyclic graph (DAG) has at least one valid topological sort. If we assume edges between vertices indicate dependencies, then a topological sorting provides an ordering of vertices such that all of the dependencies are visited before moving on to the dependents. A topological sorting is not necessarily unique – that is, there are often multiple valid topologically sorted orders for a given DAG. Here's an example below:



The graph shown to the left has many valid topological sorts, including:

- 5, 7, 3, 11, 8, 2, 9, 10 (visual left-to-right, top-to-bottom)
- 3, 5, 7, 8, 11, 2, 9, 10 (smallest-numbered available vertex first)
- 5, 7, 3, 8, 11, 10, 9, 2 (fewest edges first)
- 7, 5, 11, 3, 10, 8, 9, 2 (largest-numbered available vertex first)
- 5, 7, 11, 2, 3, 8, 9, 10 (attempting top-to-bottom, left-to-right)
- 3, 7, 8, 5, 11, 10, 2, 9 (arbitrary)

Figure 12: Valid topologically sorted orderings for the graph (from Wikipedia, https://en.wikipedia.org/wiki/Topological_sorting#Application_to_shortest_path_finding)

A topological sorting algorithm is provided in the accompanying *graph-algorithms.pdf* and shown below in Fig. 12. This algorithm is abstract and uses Mathmodels features to describe the topological sort. You must adapt this algorithm to be executable within the LIST_GRAPH class, using features available in LIST_GRAPH.

```
-- input
 graph: COMPARABLE_GRAPH [V → COMPARABLE]
-- output
topological_order: SEQ[V] -- on graph

topological_sort
  require
    graph.is_acyclic
  local
    in_degree: FUN[V, INTEGER]
      -- in_degree[v] is the number of unvisited incoming edges of v
    queue: QUEUE[V]
      -- elements having in_degree = 0, i.e. no dependencies
    front: V
  do
    from -- initialize data structures
      create queue.make_empty
      -- for every vertex compute an initial in_degree
      create in_degree.make_empty
      across graph.vertices is v loop
        in_degree := in_degree @<+ [v, graph.in_degree_count (v)]
      end
      -- if in_degree[v] = 0, then enqueue(v)
      across graph.vertices is v loop
        if in_degree[v] = 0 then
          queue.enqueue (v)
        end
      end
    until
      queue.is_empty
    loop
      front := queue.first
      queue.dequeue -- remove front of queue
      topological_order := topological_order |-> front -- append front
      -- for each adjacent edge front -> u, update in_degree:
      across graph.adjacent(front) is u loop
        in_degree := in_degree @<+ [u, in_degree[u] - 1]
        if in_degree[u] = 0 then
          queue.enqueue(u)
        end
      end
    end
  ensure
      graph.is_topologically_sorted (topological_order)
  end
```

Figure 13: Abstract Topological Sorting algorithm from graph-algorithms.pdf.

3.1 Testing class LIST_GRAPH

The test classes contain many different features that each test a different aspect of your implementation. In each case, class VERTEX is used in combination with EDGE to construct LIST_GRAPH structures.

Consider the following graph structure built using LIST_GRAPH [G]:

```
i_a := <<[1, 2], [1, 3], [3, 4], [3, 5], [5, 6]>>
i_a.compare_objects
create i_g.make_from_array (i_a)
assert_equal ("correct vertices & edges", "[1:2,3][2][3:4,5][4][5:6][6]", i_g.out)
Result := i_g.edge_count ~ 5 and i_g.vertex_count ~ 6
```

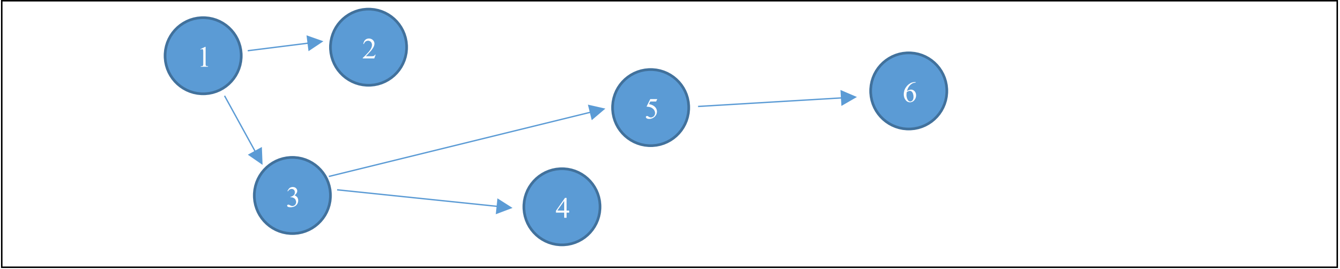


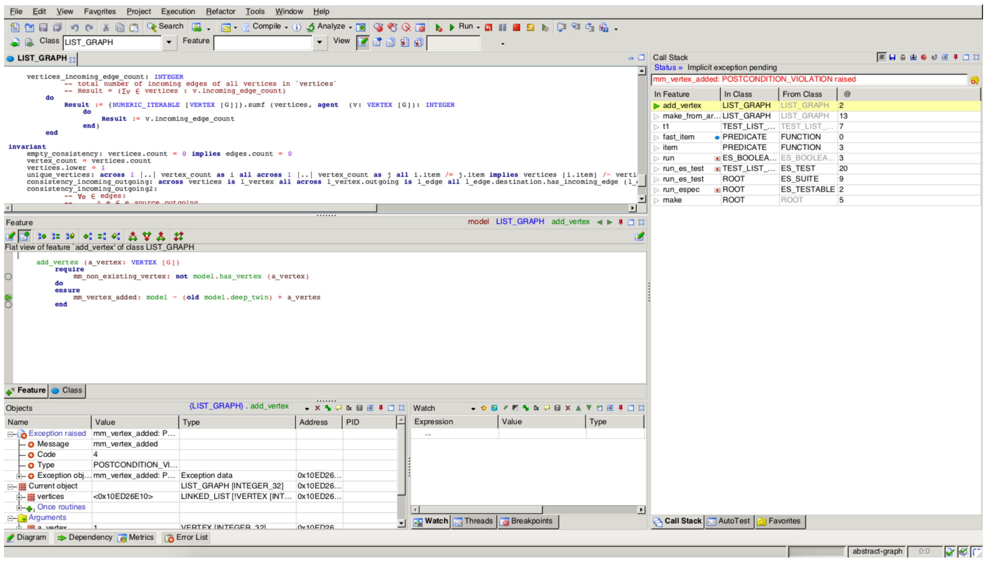
Figure 14 Test graph of integers

The test class uses the {LIST_GRAPH}.out query to ensure that the graph has inserted the vertices and edges into the graph and prints it out in the correct order.

For the *topologically_sorted* tests, it is important to note that these will be evaluated for correctness against your *is_topologically_sorted* query, which is also tested on its own. Therefore, it is important to ensure your *is_topologically_sorted* query is working correctly before you move on to *topologically_sorted*.

3.2 Initially Tests Fail

If you execute the tests (F5) then the debugger will halt at a failing test as shown below. You must learn how to use the debugger (see Eiffel-101).


File Edit View Favorites Project Execution Refactor Tools Window Help
Search Compile Analyze Run
Class LIST_GRAPH Feature View
LIST_GRAPH

```
    vertices_incoming_edge_count: INTEGER
            -- total number of incoming edges of all vertices in `vertices`
            -- Result = (Σv ∈ vertices : v.incoming_edge_count)
        do
            Result := {NUMERIC_ITERABLE [VERTEX [G]]}.sumf (vertices, agent  (v: VERTEX [G]): INTEGER
                do
                    Result := v.incoming_edge_count
                end)
        end

invariant
    empty_consistency: vertices.count = 0 implies edges.count = 0
    vertex_count = vertices.count
    vertices.lower = 1
    unique_vertices: across 1 |..| vertex_count as i all across 1 |..| vertex_count as j all i.item /= j.item implies vertices [i.item] /~ verti
    consistency_incoming_outgoing: across vertices is l_vertex all across l_vertex.outgoing is l_edge all l_edge.destination.has_incoming_edge (l_
    consistency_incoming_outgoing2:
            -- ∀e ∈ edges:
```

Flat view of feature `add_vertex' of class LIST_GRAPH

```
    add_vertex (a_vertex: VERTEX [G])
        require
            mm_non_existing_vertex: not model.has_vertex (a_vertex)
        do
        ensure
            mm_vertex_added: model ~ (old model.deep_twin) + a_vertex
        end
```

Name	Value	Type	Address	PID
Exception raised	mm_vertex_added: P...			
Message	mm_vertex_added			
Code	4			
Type	POSTCONDITION_VI...			
Exception obj...	mm_vertex_added: P...	Exception data	0x10ED26...	
Current object		LIST_GRAPH [INTEGER_32]	0x10ED26...	
vertices	<0x10ED26E10>	LINKED_LIST [!VERTEX [INT...	0x10ED26...	
Once routines				
Arguments				
a_vertex	1	VERTEX [INTEGER_32]	0x10ED26	

Expression	Value	Type
...		

Watch Threads Breakpoints
Diagram Dependency Metrics Error List
Call Stack

Status = Implicit exception pending

mm_vertex_added: POSTCONDITION_VIOLATION raised

In Feature	In Class	From Class	@
add_vertex	LIST_GRAPH	LIST_GRAPH	2
make_from_ar...	LIST_GRAPH	LIST_GRAPH	13
t1	TEST_LIST_...	TEST_LIST_...	7
fast_item	PREDICATE	FUNCTION	0
item	PREDICATE	FUNCTION	3
run	ES_BOOLEA...	ES_BOOLEA...	3
run_es_test	TEST_LIST_...	ES_TEST	20
run_es_test	ROOT	ES_SUITE	9
run_espec	ROOT	ES_TESTABLE	2
make	ROOT	ROOT	5

Figure 15 Running the tests at the start

3.3 Void safety

You are required to read the section on Void Safety in Eiffel-101 to understand the keywords **attached** and **detachable**.

4 Abstraction, Design by Contract and Design Correctness

The use of generic parameters in LIST_GRAPH [G] is a form of *abstraction by parameterization*. We seek generality by allowing the same mechanism (or algorithm) to be adapted to many different contexts by providing it with information in that context.

Read the Section in Eiffel-101 on Abstraction, DbC and Information Hiding.

5 To Submit

1. Add correct implementations as specified.
2. Work incrementally one feature at a time. Run all regression tests before moving to the next feature. This will help to ensure that you have not added new bugs, and that the prior code you developed still executes correctly.
3. Add at least 5 tests of your own to STUDENT_TEST, i.e. don't just rely on our tests.
4. Don't make any changes to classes other than the ones specified.
5. Ensure that you get a green bar for all tests. Before running the tests, always freeze first.

You must make an electronic submission as described below.

1. On Prism (Linux), *eclean* your system, freeze it, and re-run all the tests to ensure that you get the green bar.
2. *eclean* your directory *abstract-graph* again to remove all EIFGENs.

Submit your Lab from the command line as follows:

```
submit 3311 Lab2 abstract-graph
```

You will be provided with some feedback. Examine your feedback carefully. Submit often and as many times as you like.

Remember

- Your code must compile and execute on the departmental Linux system (Prism) under CentOS7. That is where it must work and that is where it will be compiled and tested for correctness.
- Equip each test t with a *comment* ("t: …") clause to ensure that the ESpec testing framework and grading scripts process your tests properly. (Note that the colon ":" in

test comments is mandatory.). An improper submission will not be given a passing grade.

- The directory structure of your folder *abstract-graph* **must** be a superset of Table 1.