Abstract Graph Algorithms in Eiffel

JSO

December 30, 2019

Contents

List of Figures

1. Graph Theory (directed graphs)

A graph G is a structure $G = (V, E)$ where V is a finite set of vertices and E is a finite set of edges. To keep matters simple we will limit our summary to *directed graphs*. For example, in the graph of Fig. 1 the vertices are $\{1, 2, 3, 4, 5, 6\}$ and there are edges such as $1 \mapsto 2$ and $2 \mapsto 5$.[1] A path through the graph is a sequence of vertices, e.g. $\langle 1, 2, 5, \cdots \rangle$. In principle, in a path, vertices may appear more than once. However, often we remove cycles in a path for brevity.

A vertex v of a graph has a set of *outgoing* and *incoming* edges. A function $adjacent\ (v)$ provides a sequence of vertices adjacent (via outgoing edges) to vertex v. For example, $adjacent(1) = \langle 4, 2 \rangle$.

If the vertices of a graph are comparable (such as integers, strings etc.), then $adjacent\ (v)$ is a sequence of vertices adjacent to v in comparable order, and we may then write: $adjacent(1) = \langle 2, 4 \rangle$. A comparable order for vertices allows us to uniquely order paths, topological sorts and other properties that directed graphs might have.

We will denote such graphs as `COMPARABLE_GRAPH[V→COMPARABLE]`, where V is a generic parameter for vertices in the graph. If vertices are not comparable then we denote the graph structure as `GRAPH[V]`.

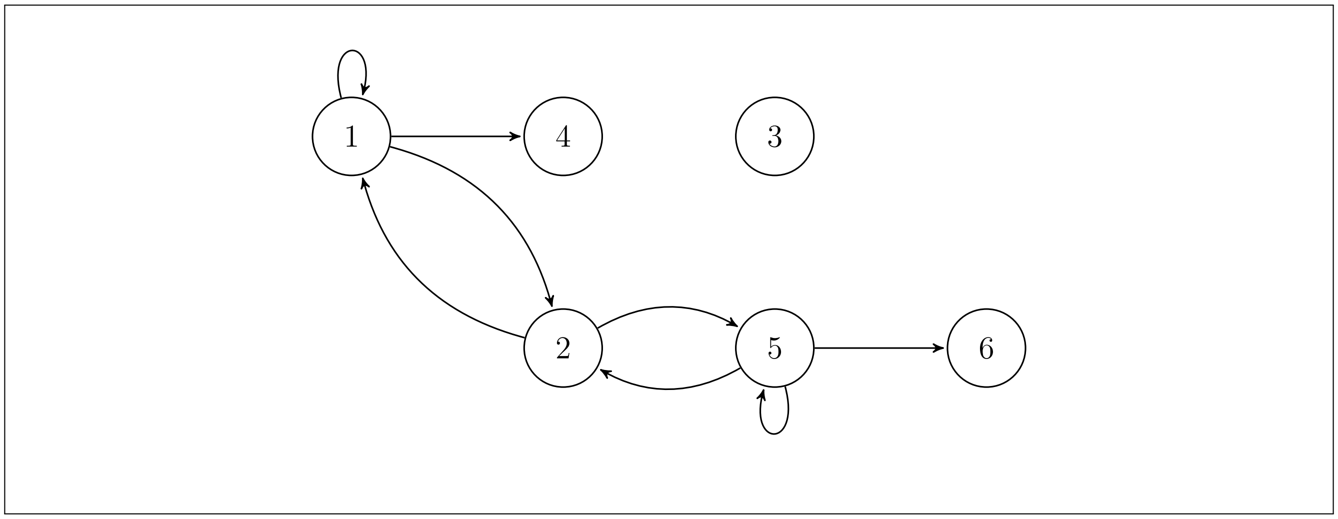

Figure 1: Directed graph with cycles and self-loops

2. Algorithms versus Programs

Mathematical entities (and algorithms based on them) are different from computer implementations of these entities. Programming languages are designed for programs that must execute efficiently on a computer. For example take something as simple as the set of integers ($\mathbb{Z}$). For efficiency, an implementation on a computer must take into account word size, so that the plus operator (for example) would have to be defined as: **IF** `x+y =< MaxInteger` **THEN** `z := x+y` **ELSE** `...` **END**; and some additional exception handling may be needed if we wish to know if overflow

[1]We do not allow parallel edges (from the same source and destination vertices)

has occurred. Operations for addition, subtraction, multiplication and quotients with 32 bit (or 64 bit) integers to take into account overflows or *NaNs* (not a number) are more complex than these operations on $\mathbb{Z}$.

To represent graphs on a computer, many implementation details are usually needed. For example, an adjacency-list representation maintains a vertex-indexed array of linked lists of the vertices connected by an edge to each vertex. In a linked list, elements are not stored at contiguous locations; the elements are linked using pointers. Or instead, an arrayed list structure might be used to store the edges. An alternative representation is via an adjacency matrix. Vertex and edge classes, pointers, iterators and many other implementation details may need to be addressed.

In describing a graph structure in section 1, we used sets, relations (sets of edge tuples), and sequences (for paths) which provide an abstract mathematical description of the graph.

When describing graph algorithms such as breadth first search, shortest path, cycle detection and topological sorts, an *abstract algorithm* is easier to describe and analyze than a *program* that implements the algorithm.

In the sequel, we thus present graph algorithms abstractly (at first) rather than as computer implementations. Of course, that abstraction must ultimately be *refined* to something that can execute on a computer. But it is easier to understand the core of the algorithm and to analyze it for efficiency and correctness without the distraction of implementation detail.

3. Algorithm for Breadth First Search (BFS)

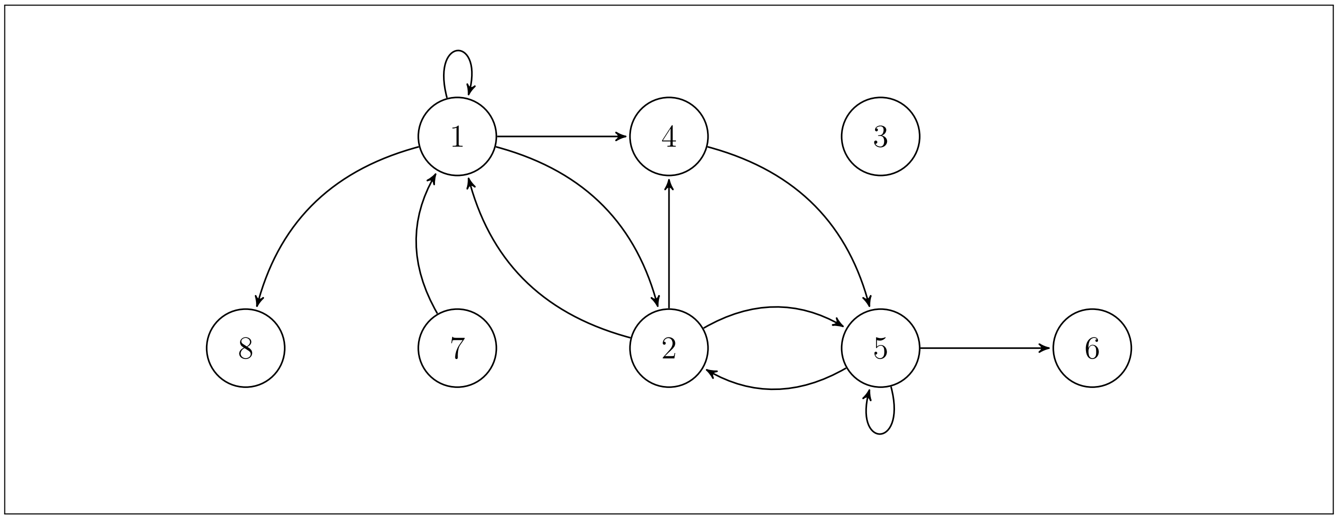

Figure 2: Shortest paths in directed graphs via BFS

```
-- inputs
  $graph$ ∈ COMPARABLE_GRAPH [V → COMPARABLE]
  $source$ ∈ V -- vertex
-- outputs
  $distance \in V \nrightarrow \mathbb{N}$ -- partial function; distance(v) is distance from source
  $parent \in V \nrightarrow V$ -- parent(v) is the parent of vertex v
  $visited \in SEQ[V]$ -- vertices already visited
    -- a sequence SEQ[V] is a special total function
    -- in $1\,..\,n \to V$ where $n$ is the size of the sequence

-- algorithm
breadth_first_search -- BFS from 'source'
  require
    $source \in graph.vertices$
  local
    $queue$: QUEUE [V] -- unvisited neighbours of visited vertices
    $front$: V
  do
    from -- initialize data structures
      $visited := \langle source \rangle$
      $distance := \{source \mapsto 0\}$
      $parent := \varnothing$
      $queue := \langle source \rangle$
    until
      $queue = \varnothing$
    loop
      $front := queue.first$;  $queue.dequeue$
      $\forall x \in graph.adjacent(front)$:
        -- $graph.adjacent(front)$ is a sequence of neighbour vertices
        if $x \notin visited$ then
          $visited := visited + x$ -- sequence append
          $distance := distance \restriction (x \mapsto distance(front) + 1)$ -- function override
          $parent := parent \restriction (x \mapsto front)$
          $queue.enqueue(x)$
        end
     end
  ensure
    $visited = graph.reachable(source)$
    $\forall v \in dom(distance) : \#graph.shortest\_path(source, v) - 1 = distance(v)$
    $\forall v \in dom(parent) : graph.shortest\_path(source, v).front.last = parent(v)$
      -- where sequence.front.last is the second last item in the sequence
    $\forall i \in 1\,..\,(\#visited - 1) : distance(visited(i)) \le distance(visited(i))$
  end

-- query
path (v: V): SEQ [V]
  -- shortest path from source to vertex v using parent function
```

Figure 3: Abstract Breadth First Search

Consider the directed graph of Fig. 2. The shortest path from a source vertex 1 to vertex 5 is the sequence $\langle 1,2,5\rangle$. There is an alternative shortest path viz. the sequence $\langle 1,4,5\rangle$. If the vertices are *comparable* (as in Fig. 2 where we use integers), then – in a breadth first search – we may uniquely denote $\langle 1,2,5\rangle$ as the shortest path (as the vertex 2 precedes vertex 4). Note that the sequence $\langle 1,2,4,5\rangle$ is also a path, but not the shortest.

The inputs for a BFS algorithm is a graph and a *source* vertex in the graph. The precondition for the BFS is thus $source \in graph.vertices$. An algorithm for breadth first search (BFS) is shown in Fig. 3 using mathematical sets, functions, and sequences. An algorithm should:

- List the inputs (and their types)
- List the expected outputs (and their types)
- Describe the algorithm
- Provide a precondition on the inputs and a postcondition on the outputs. Without a postcondition we do not know what the expected output should be. These contracts can be specified informally, or formally.

The output of our BFS algorithm provides the *visited* nodes, the shortest path, and a *parent* function on the shortest path. Usually the output *visited* would be of type `SET[V]`; however we use `SEQ[V]` instead, so that the sequence can be provided in a comparable order.

- The distance function $distance \in V \nrightarrow \mathbb{N}$ is a partial function where $distance(v)$ provides the distance from the *source* to vertex v. If there is no path from the source to v then v is not in the domain of the function. In Fig. 2, $distance(5) = 2$ and $distance(6) = 3$. Thus: $distance = \{1 \mapsto 0, 2 \mapsto 1, 4 \mapsto 1, 8 \mapsto 1, 5 \mapsto 2, 6 \mapsto 3\}$.
- Likewise the parent function $parent \in V \nrightarrow V$ is a partial function where $parent(v)$ is the parent of vertex v on the shortest path. For example, in In Fig. 2, $parent(5) = 2$. Thus $parent = \{2 \mapsto 1, 4 \mapsto 1, 8 \mapsto 1, 5 \mapsto 2, 6 \mapsto 5\}$.
- The visited nodes is the sequence $\langle 1,2,4,8,5,6\rangle$ in comparable BFS order. In the first phase, from source vertex 1, we visit 2, 4, 8. In the second phase, we visit 5. In the last phase we visit 6.
- We may also wish to support a query `path (v:V): SEQ[V]`, using the *parent* function.

4. Mathmodels and BFS

Fig. 3 provides an algorithm for breadth first search. This algorithm is abstract as it is described without distracting implementation details. It us thus a specification for any computer representation. However, it cannot be checked for type correctness nor is it execuatble.

In some ways, formal specification languages look remarkably like programming languages. To be usable for significant applications they must meet the same challenges: defining a coherent type system, supporting abstraction, providing good syntax (clear to human readers and parsable by tools), specifying the semantics, offering modular structures, allowing evolution while ensuring compatibility. Bertrand Meyer has argued that suitable object-oriented languages can help on both sides. He writes: “Eiffel as a language is the notation that attempts to support a seamless, continuous process, providing tools to express both abstract specifications and detailed implementations. One of the principal arguments for this approach is that it supports change and reuse. If

```
-- inputs
  graph: COMPARABLE_GRAPH [V → COMPARABLE]
  source: V
-- outputs
  distance: FUN [V, INTEGER]
    -- distance[v] is distance of vertex v from source
  parent: FUN [V, V]
    -- parent[v] is parent of vertex v
  visited: SEQ [V]

-- command
breadth_first_search -- from ‘source’
  require
    graph.has_vertex (a_source)
  local
    queue: QUEUE [V];
    front: V
  do
    from -- initialize data structures
      visited := <<source>>; distance := <<[source, 0]>>
      create parent.make_empty; queue := <<source>>
    until queue.is_empty
    loop
      front := queue.first; queue.dequeue
      across graph.adjacent (front) is x loop
        if not visited.has (x) then
          visited := visited + x
          distance := distance @<+ [x, distance[front]+1]
          parent := parent @<+ [x, front] -- "@<+" is function override ↾
          queue.enqueue (x)
        end
     end
  ensure -- correct outputs see: Fig.3
  end
```

Figure 4: Mathmodels: Abstract Executable Breadth First Search

everything could be fixed from the start, maybe it could be acceptable to switch notations between specification and implementation. But in practice specifications change and programs change, and a seamless process relying on a single notation makes it possible to go back and forth between levels of abstraction without having to perform repeated translations between levels. (This problem of change is, in my experience, the biggest obstacle to refinement-based approaches. I have never seen a convincing description of how one can accommodate specification changes in such a framework without repeating the whole process. Inheritance, by the way, addresses this matter much better)”.[2]

In [1] and the appendix of this paper, we describe the Eiffel Mathmodels library which consists

[2]https://bertrandmeyer.com/2014/12/07/lampsort/

of mathematical sets, sequences, functions, relations, bags, etc. This library also has a graph abstraction. The algorithm for breadth first search shown in Fig. 3 is not executable, nor can it be checked for type correctness.

With the graph ADT of Mathmodels, we may provide an executable version of breadth first search shown in Fig. 4 which does not lose much of the abstraction of Fig. 3. The graph representation in the test is very close to the mathematical model of graphs described in Section 1. We this obtain the powerful ability to provide an abstract specification of graph algorithms, to mechanically check them for type correctness and to execute them on actual graphs.

```
test: BOOLEAN
  local
    g: COMPARABLE_GRAPH[INTEGER]
    a: ARRAY[TUPLE[INTEGER, INTEGER]]
    ga: GRAPH_ALGORITHMS[INTEGER]
    shortest_path: SEQ[INTEGER]
  do
    comment ("breadth first search of graph algorithms")
    a := <<[1, 1], [1, 8], [4, 5], [1, 4], [1, 2], [2, 5], [2, 4], [5, 2], [2,
       1], [5,5], [5, 6]>>
    a.compare_objects
    create g.make_from_tuple_array (a)
    g.vertex_extend (3)
    create ga.make (g)
    ga.breadth_first_search(1)
    assert_equal ("graph correct",
        "[1:1,2,4,8][2:1,4,5][3][4:5][5:2,5,6][6][8]",
        g.debug_output)
    Result := ga.distance[1] = 0
    check Result end
    shortest_path := <<1, 2, 5>>
    Result := ga.path (5) ~ shortest_path
    assert_equal ("distance correct",
      "{ 1 -> 0, 2 -> 1, 4 -> 1, 8 -> 1, 5 -> 2, 6 -> 3 }",
      ga.distance.out)
    assert_equal ("parent correct",
      "{ 2 -> 1, 4 -> 1, 8 -> 1, 5 -> 2, 6 -> 5 }",
      ga.parent.out)
    assert_equal ("visited_correct",
      "< 1, 2, 4, 8, 5, 6 >",
      ga.visited.out)
  end
```

Figure 5: Testing the abstract BFS algorithm of Fig. 4 for graph in Fig. 2

5. Algorithm for Topological Sort

```
-- 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 6: Mathmodels: Abstract Executable Topological Sort

```
class SET [G] create
  make_empty, make_from_array
feature -- Commands with efficient implementation
  extend (g: G)
    -- extend current set by g
  union (other: like Current)
    -- union of this set with other'
  subtract (g: G)
   -- subtract this set by g
  difference (other: like Current)
    -- difference of this set and other
feature -- Immutable Queries
  count alias "#": INTEGER
    -- cardinality of this set
  is_empty: BOOLEAN
    -- is this set empty?
  has (g: G): BOOLEAN
    -- does this set contain g?
  extended alias "+" (g: G): like Current
    -- a new set representing the addition of g to this set
  unioned alias "|\/|" (other: like Current): like Current
    -- a new set representing the union of other to this set
  subtracted alias "-" (other: like Current): like Current
    -- a new set representing the subtraction of g from this set
  differenced alias "|\'" (other: like Current): like Current
    -- a new set representing the difference between this and other
  ...
end
```

Figure 7: API for class `SET[G]` from Mathmodels with immutable queries

```
class GRAPH[V] create
  make_empty, make_from_tupled_array
feature -- immutable queries
  vertices: SET [V]
  edges: REL [V, V]
  incoming (v: V): REL [V, V]
  outgoing (v: V): REL [V, V]
  adjacent (v: V): SEQ [V]
  has_edge (p: PAIR [V, V]): BOOLEAN
  has_vertex (v: V): BOOLEAN
  edge_extended alias "|\/|" (e: PAIR [V, V]): GRAPH [V]
  edge_removed alias "|\" (e: PAIR [V, V]): GRAPH [V]
  is_a_shortest_path (v1, v2: V; seq: SEQ [V]): BOOLEAN
  is_acyclic: BOOLEAN
  is_empty: BOOLEAN
  is_topologically_sorted (seq: SEQ [V]): BOOLEAN
  ...
feature -- commands
   edge_extend (e: TUPLE [V, V])
   edge_remove (e: TUPLE [V, V])
   vertex_extend (v: V)
   vertex_remove (v: V)
end
```

Figure 8: API for class `GRAPH[G]` from Mathmodels with immutable queries

A. Mathmodels Library for Specifications

```
class REL [G, H] inherit
 SET[TUPLE[G, H]]
create
 make_empty
feature -- queries
 domain: SET [G]
  -- Return the domain set of relation.
 range: SET [H]
  -- Return the range set of relation.
 image alias [] (g: G): SET [H]
  -- Retrieve set of range items
  -- for domain element g
  extended alias "+" (p:TUPLE[G, H]): REL [G, H]
    -- return a new relation with addition of t
  overriden_by (p: TUPLE[G, H]): REL [G, H]
   -- Return a new relation the same as Current,
   -- except p.first now maps to p.second
   -- alias ``@<+''

   ...
end
```

Figure 9: Some Queries of Mathmodels REL[G, H]

The *Mathmodels* library helps to improve seamlessness and reversibility in the development of reliable software. It has classes for functions, sequences, relations (see Table 9), bags etc. It is used for model-based specification—an approach to specifying software components where the system specification is expressed as a state model.

A.1. Abstract State Machines

A well-known approach to formal specification is algebraic specification. A stack in generic parameter G is described as an abstract datatype having operations (mathematical functions or partial functions) and axioms. The axioms allow us to deduce the behaviour of the stack free of implementation detail. An object-oriented class may implement this datatype. Consider the Eiffel class STACK whose interface (contracts but no implementation) is shown in Table 1.

Without the Mathmodels library, classical contracting only partially specifies the stack behaviour. Consider the feature *pop* in the stack class (ignoring, for the moment, the query *model*).[3] The precondition is classical because it can be written using the queries (in this case *count*). Classically, we cannot write a meaningful postcondition at the abstract level of a specification.

[3] In Eiffel, a feature is either a *command* routine or a *query*. A command changes the state of the current object, while queries have no side effects on the current object but return a value. A query can either be an attribute (a variable) or a function routine.

Table 1: An Eiffel Abstract State Machine for a Stack

```
class STACK [G] create
 make

feature -- model
 model: SEQ [G] -- abstraction function

feature -- queries
 count: INTEGER -- number of items in stack
   ensure Result = model.count

 top: G -- top of stack
   require count > 0
   ensure Result ∼ model [1]

feature -- commands
 pop -- pop top of stack
   require count > 0
   ensure model ∼ old model.tail

 push (x: G) -- push 'x' on to the stack
   ensure model ∼ (x ◁ old model)

invariant
    model.count = implementation.count
    -- implementation not shown
end
```

Once an implementation for the stack is provided (e.g. a linked list), then we can provide complete contracts in terms of the implementation. But these implementation contracts are "polluted" with code detail (such as the previous link of a node) that are irrelevant to the abstract specification.

To provide a complete abstract specification, we may use the Mathmodels library sequence class `SEQ[G]` to provide an abstract state model for a stack. We do this by declaring a query *model* of type `SEQ[G]`. Then the postcondition of *pop* asserts that the new *model* is equal to the tail of the old model sequence. Likewise the postcondition of *push* is the old *model* prepended with the new item x.[4]

[4]In the Table, for simplicity of presentation, we write the postcondition as `model ∼ x ◁ old model)`. In Eiffel, the postcondition is written in ASCII:
`model ∼ old model.deep_twin |<- x`
where ∼ is the symbol for object equality. We take the deep twin of the old model to ensure that there is no issue with aliasing (omitted for brevity in the Table). Eiffel syntax allows for an infix operator "`|<-`" (in ASCII) as an alias for the prepended query in class `SEQ`. In Eiffel, given two reference variables `v1` and `v2`, `v1 = v2` is used for reference equality and `v1∼v2` is the Eiffel symbol for object equality. In Mathmodel, we use a value semantics, and thus always compare objects using object equality.

Query *prepended* (with infix notation "$\lhd$") in Table 1 is a function routine that returns a new sequence the same as the old one prepended with the argument x. It has no side effects on the current model sequence of the stack. The queries of Mathmodel classes such as SEQ act like mathematical functions. Hence they may safely be used in contracts following the Command-Query separation principle.[5]

A.2. Abstraction Function

```
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 10: Abstraction function: *model*

When specifying the stack, query *model* may be an attribute (not needing any implementation), and all contracts are specified in terms of the *model* sequence.

Class STACK may be implemented in a variety of ways, e.g. with an array or a linked list. Once an implementation for the stack is chosen, then query *model* must be given a body that maps the the concrete implementation state (in terms of an array or linked list) into into a sequence representing the abstract state as shown in Table 10 where the implementation of the stack is in terms of an array. Whatever the implementation, the body of query routine *model* must return a sequence equivalent to the implementation. The command prepend Mathmodel class SEQ is used to deduce the model sequence.

In subsection A.4 below we describe the difference between commands such as prepend of the Mathmodel class SEQ used in the body of the model function routine in Table 10 and queries such as prepended.

A.3. Specifications vs. Implementations

A software specification normally describes the set of services a system or component is expected to provide (e.g. *push*, *pop* and *top* in the simple case of the stack). It must be be precise so that it can act as a contract between the client and the supplier (understandable by both). A specification is an abstraction. It should describe the important aspects and omit the unimportant ones. The

[5]Queries return a result and do not change the observable state of the system (are free of side effects). Commands change the state of a system but do not return a value.

specification should describe everything one needs to know to use the code. It should never be necessary to read the code to find out what it does. Specifications allow us to think above the code level so as to understand our programming task at a higher level before we start writing code. Thus a specification describes what the system will do but not how it will do it. The Mathmodel contracts in Table 1 are thus a specification of the stack.

By contrast, the stack can be implemented in many different ways. In Table 10, the stack is implemented by an array. But it may also be implemented by a linked list or other data structures. Implementations thus describe how the stack will perform its operations. Implementations can change, but the Mathmodel specifications remain the same. This is the power of abstraction.

The stack example in Table 1 provides a simple illustration of what we mean by seamlessness. Formal specification languages must meet the same challenges as programming languages such as defining a coherent type system, supporting abstraction and modularity, and providing a clear syntax and semantics. In the stack, we use the same notation (Eiffel in this case) to express specifications and implementations within the same syntactic and semantic universe. In an ideal world where requirements are fixed at the start, one might switch notations between specification and implementation. But in practice requirements, designs and implementations change, and a seamless process relying on a single wide spectrum notation makes it possible to go back and forth between levels of abstraction without having to perform repeated translations between levels.

Thus, should we change the implementation of the STACK to a linked list, then all the Mathmodel contracts remain the same. Only the abstraction function `model` must be changed to reflect the new mapping from concrete to abstract state. If all unit tests are written at the interface level in terms of the public features, then the tests also remain unchanged despite the change in implementation. Contract violations at runtime will signal inconsistencies between specifications and implementations.

A.4. Immutable Functions vs. Commands

Once an implementation for the stack is chosen, then function routine *model* must be given a body that maps the implementation into into a sequence representing the abstract state. If the implementation changes, then the body of *model* must be changed to reflect the new abstraction function, but all other contracts remain unchanged. The mathematical class `SEQ` itself contains:

- Side-effect free function routines such as *prepended* (infix notation "$\lhd$") used in contracts as shown in Table 1;
- Command routines such as *prepend* that change the state of the current sequence as shown in the body of *model* in Table 10.

The commands of `SEQ` can also be used to implement the stack, although perhaps not as efficiently as standard arrays, lists, dictionaries etc..

The feature *prepend* (of class `SEQ`) is a command that changes the state of its target. The call `Result.prepend(argument)`, in the body of query *model* in Table 10, is a command that changes the state of the result. Queries of Mathmodel classes have command analogues. Thus a Mathmodel class such as `SEQ` can be used for specifications (via its queries) and (high-level but relatively) inefficient implementations (via its commands), that can later be refined to more efficient code without changing the specifications.

We specify the behaviour of the stack as an abstract state machine (using a mathematical sequence for the state) rather than as an abstract datatype with axioms to define behaviour. The idea of specifying a system by writing down all its axioms (as in abstract datatypes) seems like a good approach. But in practice, it often hard to decide whether the specification is complete or what additional properties are or are not implied by the axioms. Thus, specifying systems using abstract state machines (as in Table 1) have become more popular. In the case of languages that support DbC, we obtain the benefit of specifications and final implementation as efficient executable code in a modern object oriented language that can be tested to satisfy the specification.

References

[1] Jonathan S. Ostroff and Chen-wei Wang. Modelling and testing requirements via executable abstract state machines. In *2018 IEEE 8th International Model-Driven Requirements Engineering Workshop (MoDRE)*, pages 1–10, 2018.