UNIVERSITY OF Southampton

Towards Verified Implementation of Event-B Models in Dafny

MohammadSadegh Dalvandi, Michael Butler

5th Rodin User and Developer Workshop, June 2014, Toulouse

Software Correctness: Constructive & Analytical Approaches

- **Constructive Approach:** Verification of formal models of required system behaviour at different levels of abstraction

- *Aims at early stages of development*

- **Analytical Approach:** Verification of properties of program code against formal specification of the system

- *Aims at coding stage*

Linking Constructive & Analytical Approaches

- Constructive and analytical approaches are *mutually beneficial*
- Constructive approach for verifying *high-level functional properties*
- Analytical approach for verifying *implementation-oriented properties*
- To investigate this, we are trying to build a link between these two approaches.

Linking Event-B and Dafny

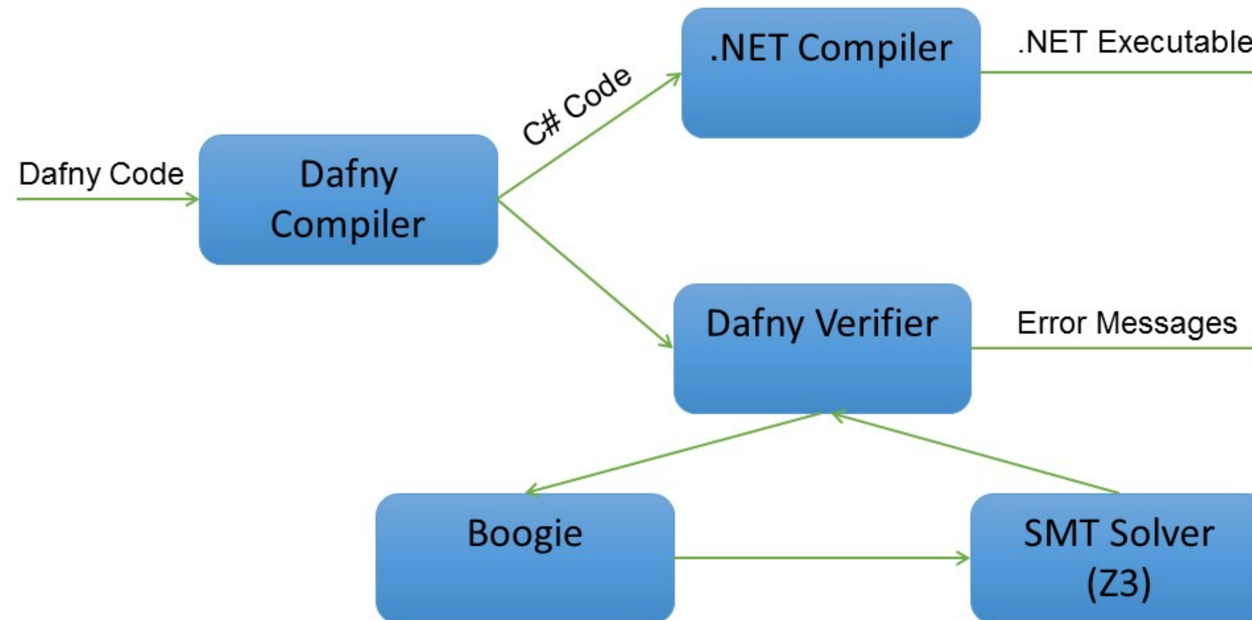
Event-B

- A formal method for system-level modelling and analysis
- Based on set theory and predicate logic
- Supporting refinement
- *Constructive* approach

Dafny

- A language and program verifier for functional correctness
- Has built-in specification constructs: pre- & post-conditions, assertion,…
- *Analytical* approach

Dafny: A Language and Program Verifier



Case Study: A Map Abstract Data Type (1)

Dafny:

- Specification: using two sequences *keys* and *values*
- Implementation: using a linked-list where each node has three fields: *key, val,* and *next*

Event-B:

- Map modelled as *partial function* in the most abstract level
- *Sequences* and *linked-list* added through refinement levels
- Event-B extended by *Theory Plug-in* to accommodate sequences

Case Study: A Map Abstract Data Type (2)

- Adding a new key to the map (Dafny implementation):

```
method Add(key: Key, val: Value)
  requires Valid();
  modifies Repr;
  ensures Valid() && fresh(Repr - old(Repr));
  ensures forall i :: 0 <= i < |old(Keys)| && old(Keys)[i] == key ==>
              |Keys| == |old(Keys)| &&
              Keys[i] == key && Values[i] == val &&
              (forall j :: 0 <= j < |Values| && i != j ==>
                  Keys[j] == old(Keys)[j] && Values[j] == old(Values)[j]);
  ensures key !in old(Keys) ==> Keys == [key] + old(Keys) && Values == [val] + old(Values);
{
  var p, n, prev := FindIndex(key);
  if (p == null) {
    var h := new Node<Key,Value>;
    h.key := key;  h.val := val;  h.next := head;
    head := h;
    Keys := [key] + Keys;  Values := [val] + Values;
    nodes := [h] + nodes;
    Repr := Repr + {h};
  } else {
    p.val := val;
    Values := Values[n := val];
  }
}
```

Method Specification

Method Body

Case Study: A Map Abstract Data Type (2)

- Adding a new key to the map (Dafny implementation):

```
method Add(key: Key, val: Value)
  requires Valid();
  modifies Repr;
  ensures Valid() && fresh(Repr - old(Repr));
  ensures forall i :: 0 <= i < |old(Keys)| && old(Keys)[i] == key ==>
            |Keys| == |old(Keys)| &&
            Keys[i] == key && Values[i] == val &&
            (forall j :: 0 <= j < |Values| && i != j ==>
                Keys[j] == old(Keys)[j] && Values[j] == old(Values)[j]);
  ensures key !in old(Keys) ==> Keys == [key] + old(Keys) && Values == [val] + old(Values);
{
  var p, n, prev := FindIndex(key);
  if (p == null) {
    var h := new Node<Key,Value>;
    h.key := key;  h.val := val;  h.next := head;
    head := h;
    Keys := [key] + Keys;  Values := [val] + Values;
    nodes := [h] + nodes;
    Repr := Repr + {h};
  } else {
    p.val := val;
    Values := Values[n := val];
  }
}
```

If *key* is already exists in the map, returns the node that *key* is stored in it, the position of the node in linked-list, and the previous node in the list. Otherwise returns *null*

Adding a new *key* to the map

Updating the value of an existing *key*

Case Study: A Map Abstract Data Type (3)

- Event-B model for adding new keys to the map(first refinement):

```
Add1  ≙
 extended
   STATUS
 ordinary
REFINES
 Add
ANY
 k
 v
WHERE
 grd1  :  k∈KEYS
 grd2  :  v∈VALUES
 grd3  :  k∉dom(map)
 grd4  :  k∉ran(keys)
THEN
 act1  :  map(k)≔v
 act2  :  keys≔seqPrepend(keys,k)
 act3  :  values≔seqPrepend(values,v)
 act4  :  n≔n+1
END
```

```
method Add(key: Key, val: Value)
  requires Valid();
  modifies Repr;
  ensures Valid() && fresh(Repr - old(Repr));
  ensures forall i :: 0 <= i < |old(Keys)| && old(Keys)[i] == key ==>
              |Keys| == |old(Keys)| &&
              Keys[i] == key && Values[i] == val &&
              (forall j :: 0 <= j < |Values| && i != j ==>
                  Keys[j] == old(Keys)[j] && Values[j] == old(Values)[j]);
  ensures key !in old(Keys) ==> Keys == [key] + old(Keys) && Values == [val] + old(Values);
{
  var p, n, prev := FindIndex(key);
  if (p == null) {
    var h := new Node<Key,Value>;
    h.key := key;  h.val := val;  h.next := head;
    head := h;
    Keys := [key] + Keys;  Values := [val] + Values;
    nodes := [h] + nodes;
    Repr := Repr + {h};
  } else {
    p.val := val;
    Values := Values[n := val];
  }
}
```

Case Study: A Map Abstract Data Type (4)

- Event-B model for updating value of existing keys (first refinement):

```
Add2  ≙
 extended
   STATUS
 ordinary
REFINES
 Add
ANY
 k
 v
 i
WHERE
 grd1  :  k∈KEYS
 grd2  :  v∈VALUES
 grd3  :  i∈1‥n
 grd4  :  keys(i)=k
THEN
 act1  :  map(k)≔v
 act2  :  values(i)≔v
END
```

```
method Add(key: Key, val: Value)
  requires Valid();
  modifies Repr;
  ensures Valid() && fresh(Repr - old(Repr));
  ensures forall i :: 0 <= i < |old(Keys)| && old(Keys)[i] == key ==>
              |Keys| == |old(Keys)| &&
              Keys[i] == key && Values[i] == val &&
              (forall j :: 0 <= j < |Values| && i != j ==>
                  Keys[j] == old(Keys)[j] && Values[j] == old(Values)[j]);
  ensures key !in old(Keys) ==> Keys == [key] + old(Keys) && Values == [val] + old(Values);
{
  var p, n, prev := FindIndex(key);
  if (p == null) {
    var h := new Node<Key,Value>;
    h.key := key;  h.val := val;  h.next := head;
    head := h;
    Keys := [key] + Keys;  Values := [val] + Values;
    nodes := [h] + nodes;
    Repr := Repr + {h};
  } else {
    p.val := val;
    Values := Values[n := val];
  }
}
```

Case Study: A Map Abstract Data Type (5)

- Event-B model for adding new keys (second refinement):

```
Add1   ≙          //   Add new KEY (and V.
 extended
   STATUS
 ordinary
REFINES
 Add1
ANY
 k
 v
 nod
WHERE
 grd1   :    k∈KEYS
 grd2   :    v∈VALUES
 grd3   :    k∉dom(map)
 grd4   :    k∉ran(keys)
 grd5   :    nod∈NODE
 grd6   :    nod∉node
 grd7   :    nod≠null
 grd8   :    nod∉ran(nodes)
THEN
 act1   :    map(k)≔v
 act2   :    keys≔seqPrepend(keys,k)
 act3   :    values≔seqPrepend(values,v)
 act4   :    n≔n+1
 act5   :    node≔node∪{nod}
 act6   :    key(nod)≔k
 act7   :    val(nod)≔v
 act8   :    next≔next∪{nod↦head}
 act9   :    nodes≔seqPrepend(nodes,nod)
 act10   :    head≔nod
END
```

```
method Add(key: Key, val: Value)
  requires Valid();
  modifies Repr;
  ensures Valid() && fresh(Repr - old(Repr));
  ensures forall i :: 0 <= i < |old(Keys)| && old(Keys)[i] == key ==>
              |Keys| == |old(Keys)| &&
              Keys[i] == key && Values[i] == val &&
              (forall j :: 0 <= j < |Values| && i != j ==>
                  Keys[j] == old(Keys)[j] && Values[j] == old(Values)[j]);
  ensures key !in old(Keys) ==> Keys == [key] + old(Keys) && Values == [val] + old(Values);
{
  var p, n, prev := FindIndex(key);
  if (p == null) {
    var h := new Node<Key,Value>;
    h.key := key;  h.val := val;  h.next := head;
    head := h;
    Keys := [key] + Keys;  Values := [val] + Values;
    nodes := [h] + nodes;
    Repr := Repr + {h};
  } else {
    p.val := val;
    Values := Values[n := val];
  }
}
```

Case Study: A Map Abstract Data Type (6)

- Event-B model for updating value of existing keys (second refinement):

```
Add2 ≙
  extended
    STATUS
  ordinary
REFINES
  Add2
ANY
  k
  v
  i
  nod
WHERE
  grd1  :  k∈KEYS
  grd2  :  v∈VALUES
  grd3  :  i∈1‥n
  grd4  :  keys(i)=k
  grd5  :  nod∈node
  grd6  :  nodes(i)=nod
  grd7  :  key(nod)=k
THEN
  act1  :  map(k)≔v
  act2  :  values(i)≔v
  act3  :  val(nod)≔v
END
```

```
method Add(key: Key, val: Value)
  requires Valid();
  modifies Repr;
  ensures Valid() && fresh(Repr - old(Repr));
  ensures forall i :: 0 <= i < |old(Keys)| && old(Keys)[i] == key ==>
                |Keys| == |old(Keys)| &&
                Keys[i] == key && Values[i] == val &&
                (forall j :: 0 <= j < |Values| && i != j ==>
                    Keys[j] == old(Keys)[j] && Values[j] == old(Values)[j]);
  ensures key !in old(Keys) ==> Keys == [key] + old(Keys) && Values == [val] + old(Values);
{
  var p, n, prev := FindIndex(key);
  if (p == null) {
    var h := new Node<Key,Value>;
    h.key := key;  h.val := val;  h.next := head;
    head := h;
    Keys := [key] + Keys;  Values := [val] + Values;
    nodes := [h] + nodes;
    Repr := Repr + {h};
  } else {
    p.val := val;
    Values := Values[n := val];
  }
}
```

Case Study: A Map Abstract Data Type (7)

```
Add2 ≙                // Ec
  extended
    STATUS
  ordinary
REFINES
  Add2
ANY
  k
  v
  i
  nod
WHERE
  grd1  :  k∈KEYS
  grd2  :  v∈VALUES
  grd3  :  i∈1··n
  grd4  :  keys(i)=k
  grd5  :  nod∈node
  grd6  :  nodes(i)=nod
  grd7  :  key(nod)=k
THEN
  act1  :  map(k)≔v
  act2  :  values(i)≔v
  act3  :  val(nod)≔v
END
```

```
method Add(key: Key, val: Value)
  requires Valid();
  modifies Repr;
  ensures Valid() && fre
  ensures forall i :: 0                                        ==>
              |Keys| ==
              Keys[i] =
              (forall j :
                  Keys[j] ==                                  Values)[j]);
  ensures key !in old(Keys) ==> Ke                        && Values == [val] + old(Values);
{
  var p, n, prev := FindIndex(key);
  if (p == null) {
    var h := new Node<Key,Value>;
    h.key := key;  h.val := val;  h.next := head;
    head := h;
    Keys := [key] + Keys;  Values := [val] + Values;
    nodes := [h] + nodes;
    Repr := Repr + {h};
  } else {
    p.val := val;
    Values := Values[n := val];
  }
}
```

FindIndex(key) is implicitly modelled as guards of events *Add1* & *Add2*

Case Study: A Map Abstract Data Type (8)

- Invariants:
  - Dafny does not have any special construct for invariants
  - Invariants are placed in a boolean function called *Valid()*
  - *Valid()* is a pre- and post-condition for all methods

```
function Valid(): bool
  reads this, Repr;
{
  this in Repr &&
  |Keys| == |Values| && |nodes| == |Keys| + 1 &&
  head == nodes[0] &&
  (forall i :: 0 <= i < |Keys| ==>
      nodes[i] != null &&
      nodes[i] in Repr &&
      nodes[i].key == Keys[i] && nodes[i].key !in Keys[i+1..] &&
      nodes[i].val == Values[i] &&
      nodes[i].next == nodes[i+1]) &&
  nodes[|nodes|-1] == null
}
```

```
inv4   :  seqSize(keys)=n
inv5   :  seqSize(values)=n

inv7   :  seqSize(nodes)=seqSize(keys)+1
inv8   :  head=nodes(1)
inv9   :  ∀i·i∈1‥n ⇒ nodes(i)∈dom(key) ∧ nodes(i)∈dom(val) ∧ nodes(i)∈dom(next)
inv10  :  ∀i·i∈1‥n ⇒ nodes(i)≠null
inv11  :  ∀i·i∈1‥n ⇒ key(nodes(i))=keys(i)
inv12  :  ∀i·i∈1‥n ⇒ key(nodes(i))∉ran(seqSliceFromN(keys,i+1))
inv13  :  ∀i·i∈1‥n ⇒ val(nodes(i))=values(i)
inv14  :  ∀i·i∈1‥n ⇒ next(nodes(i))=nodes(i+1)
inv15  :  nodes(seqSize(nodes))=null
```

Thank you for your attention.

Questions? Comments?