U.PORTO
FEUP FACULDADE DE ENGENHARIA UNIVERSIDADE DO PORTO

Master in Informatics and Computing Engineering (M.EIC)

M.EIC037 | Formal Methods for Critical Systems 2021/22

Design by contract and verification of object-oriented programs in Dafny (Solutions)

1. Stack

```
type T = int // to allow doing new T[capacity], but can be other type

class {:autocontracts} Stack {
    const elems: array<T>; // immutable (pointer)
    var size : nat; // used size

    predicate Valid() {
        size <= elems.Length
    }

    constructor (capacity: nat)
      requires capacity > 0
      ensures elems.Length == capacity && size == 0
    {
        elems := new T[capacity];
        size := 0;
    }

    predicate method isEmpty() {
        size == 0
    }

    predicate method isFull() {
        size == elems.Length
    }

    method push(x : T)
      requires !isFull()
      ensures elems[..size] == old(elems[..size]) + [x]
    {
        elems[size] := x;
        size := size + 1;
    }

     function method top() : T
      requires !isEmpty()
    {
         elems[size-1]
    }

    method pop()
      requires !isEmpty()
      ensures elems[..size] == old(elems[..size-1])
    {
         size := size-1;
    }
}
```

```
// A simple test case.
method testStack()
{
    var s := new Stack(3);
    assert s.isEmpty();
    s.push(1);
    s.push(2);
    s.push(3);
    assert s.top() == 3;
    assert s.isFull();
    s.pop();
    assert s.top() == 2;
    print "top = ", s.top(), " \n";
}
```

2. Person

```
datatype Sex = Masculine | Feminine
datatype CivilState = Single | Married | Divorced | Widow | Dead

class Person
{
    const name: string; // ‘const’ for immutable fields
    const sex: Sex;
    const mother: Person?; // ‘?’ to allow null
    const father: Person?;
    var spouse: Person?;
    var civilState: CivilState;
    ghost const ancestors : set<Person>;

    // Class invariant
    predicate Valid()
       reads this, mother, father, spouse
    {
        (spouse != null <==> this.civilState == Married)
        && (mother != null ==> mother.sex == Feminine)
        && (father != null ==> father.sex == Masculine)
        && (spouse != null ==> spouse.sex != this.sex && spouse.spouse == this
                               && spouse !in ancestors
                               && (spouse.father != null ==> spouse.father != this.father)
                               && (spouse.mother != null ==> spouse.mother != this.mother))
        && (ancestors == (if mother != null then {mother} + mother.ancestors else {})
                       + (if father != null then {father} + father.ancestors else {}))
    }

    constructor (name: string, sex: Sex, father: Person?, mother: Person?)
        // semantic constraints
        requires (mother != null ==> mother.sex == Feminine)
              && (father != null ==> father.sex == Masculine)
        // effects (equivalent to body)
        ensures this.name == name && this.sex == sex && this.mother == mother && this.father ==
father
            && this.spouse == null && this.civilState == Single
            && (this.ancestors == (if mother != null then {mother} + mother.ancestors else {})
                               + (if father != null then {father} + father.ancestors else {}))
        // validity of final objects' states
        ensures Valid()
    {
        this.name := name;
        this.sex := sex;
        this.mother := mother;
```

```
        this.father := father;
        this.spouse := null;
        this.civilState := Single;
        this.ancestors := (if mother != null then {mother} + mother.ancestors else {})
                        + (if father != null then {father} + father.ancestors else {});
    }

    method marry(spouse: Person)
        // validity of initial objects' states
        requires this.Valid() && spouse.Valid()
        // semantic constraints
        requires this.civilState !in {Married, Dead} && spouse.civilState !in {Married, Dead}
                && spouse.sex != this.sex && spouse !in ancestors && this !in spouse.ancestors
                && (spouse.father != null ==> spouse.father != this.father)
                && (spouse.mother != null ==> spouse.mother != this.mother)
        // modified objects
        modifies this, spouse
        // effects (equivalent to body)
        ensures spouse.civilState == Married && spouse.spouse == this &&
               this.civilState == Married && this.spouse == spouse
        // validity of final objects' states
        ensures this.Valid() && spouse.Valid()
    {
        spouse.spouse := this;
        spouse.civilState := Married;
        this.spouse := spouse;
        this.civilState := Married;
    }

    method divorce()
        // semantic constraints
        requires civilState == Married
        // validity of initial objects' states
        requires this.Valid() && spouse.Valid()
        // modified objects
        modifies this, spouse
        // effects (equivalent to body)
        ensures old(spouse).spouse == null && old(spouse).civilState == Divorced
                && this.spouse == null && this.civilState == Divorced
        // validity of final objects' states
        ensures this.Valid() && old(spouse).Valid()
    {
        spouse.spouse := null;
        spouse.civilState := Divorced;
        this.spouse := null;
        this.civilState := Divorced;
    }

    method die()
        // validity of initial objects' states
        requires this.Valid() && (spouse != null ==> spouse.Valid())
        // semantic constraints
        requires civilState != Dead
        // modified objects
        modifies this, spouse
        // effects (equivalent to body)
        ensures (old(spouse) != null ==> old(spouse).civilState == Widow && old(spouse).spouse
== null)
               && this.civilState == Dead && this.spouse == null
        // validity of final objects' states
        ensures this.Valid() && (old(spouse) != null ==> old(spouse).Valid())
    {
        if spouse != null
        {
            spouse.spouse := null;
            spouse.civilState := Widow;
        }
        this.civilState := Dead;
```

```
        this.spouse := null;
    }
}

// Test scenario to cover all valid transitions of civil state (checking post-conditions).
method testCivilStateTransitions()
{
    var joao  := new Person("João", Masculine, null, null);
    assert joao.spouse == null && joao.civilState == Single;
    var jose  := new Person("José", Masculine, null, null);
    var maria := new Person("Maria", Feminine, null, null);
    var luisa := new Person("Luisa", Feminine, null, null);
    joao.marry(maria); // Single -> Married
    assert joao.spouse == maria && joao.civilState == Married;
    assert maria.spouse == joao && maria.civilState == Married;
    var ana   := new Person("Ana", Feminine, joao, maria);
    joao.divorce();    // Married -> Divorced
    assert joao.spouse == null && joao.civilState == Divorced;
    assert maria.spouse == null && maria.civilState == Divorced;
    maria.die();       // Divorced -> Dead
    assert maria.spouse == null && maria.civilState == Dead;
    joao.marry(luisa); // Single, Divorced -> Married
    assert joao.spouse == luisa && joao.civilState == Married;
    assert luisa.spouse == joao && luisa.civilState == Married;
    joao.die();        // Married -> Dead, Widow
    assert joao.spouse == null && joao.civilState == Dead;
    assert luisa.spouse == null && luisa.civilState == Widow;
    luisa.marry(jose); // Single, Widow -> Married
    assert jose.spouse == luisa && jose.civilState == Married;
    assert luisa.spouse == jose && luisa.civilState == Married;
    jose.die();        // Married -> Dead, Widow
    assert jose.spouse == null && jose.civilState == Dead;
    luisa.die();       // Widow -> Dead
    assert luisa.spouse == null && luisa.civilState == Dead;
    ana.die();         // Single -> Dead
    assert ana.spouse == null && ana.civilState == Dead;
}

// Examples of invalid test scenarios (violating pre-conditions).
/*
method testInvalidParents()
{
    var joao  := new Person("João", Masculine, null, null);
    var maria := new Person("Maria", Feminine, null, null);
    var ana   := new Person("Ana", Feminine, maria, joao);
}

method testInvalidMarriage1()
{
    var maria := new Person("Maria", Feminine, null, null);
    var ana   := new Person("Ana", Feminine, null, null);
    maria.marry(ana);
}

method testInvalidMarriage2()
{
    var joao  := new Person("João", Masculine, null, null);
    var maria := new Person("Maria", Feminine, null, null);
    var ana   := new Person("Ana", Feminine, joao, maria);
    joao.marry(ana);
}

method testInvalidMarriage3()
{
    var joao  := new Person("João", Masculine, null, null);
    var maria := new Person("Maria", Feminine, null, null);
    var ana   := new Person("Ana", Feminine, null, null);
    joao.marry(maria);
```

```
    joao.marry(ana);
}

method testInvalidDivorce1()
{
    var joao  := new Person("João", Masculine, null, null);
    joao.divorce();
}

method testInvalidDivorce2()
{
    var joao  := new Person("João", Masculine, null, null);
    var maria := new Person("Maria", Feminine, null, null);
    joao.marry(maria);
    joao.die();
    maria.divorce();
}

method testInvalidDeath()
{
    var joao  := new Person("João", Masculine, null, null);
    joao.die();
    joao.die();
}
*/
```