

LinkedIn Network

Formal Modeling | VDM++

Mestrado Integrado em Engenharia Informática e Computação

Métodos Formais em Engenharia de Software

*Something something, a name here?*

Contents

[1. Informal system description and list of requirements 3](#_Toc435377798)

[1.1 Informal system description 3](#_Toc435377799)

[1.2 List of requirements 3](#_Toc435377800)

[2. Visual UML model 4](#_Toc435377801)

[2.1 Use case model 4](#_Toc435377802)

[2.2 Class model 6](#_Toc435377803)

[3. Formal VDM++ model 7](#_Toc435377804)

[3.1 Class MoneyUtils 7](#_Toc435377805)

[3.2 Class Product 8](#_Toc435377806)

[3.3 Class VendingMachine 9](#_Toc435377807)

[4. Model validation 12](#_Toc435377808)

[4.1 Class MyTestCase 12](#_Toc435377809)

[4.2 Class TestVendingMachine 12](#_Toc435377810)

[5. Model verification 16](#_Toc435377811)

[5.1 Example of domain verification 16](#_Toc435377812)

[5.2 Example of invariant verification 16](#_Toc435377813)

[6. Conclusions 17](#_Toc435377814)

[7. References 17](#_Toc435377815)

# 1. Informal system description and list of requirements

## 1.1 Informal system description –TODO // FINISH

**Location:**

**Add**

**Upload Curriculum**

Displays common connections with queried user

**Query Person: <Name\_Here>**

**Common Connections**

John, April

**Search**

**Skills**

C++; Java; …

Search for a user

**Carl Rogers**

**Connections: 42**

John, Johanna, April, …

Shows each of the user connections and total nr. of connections.



**Add**

## 1.2 List of requirements

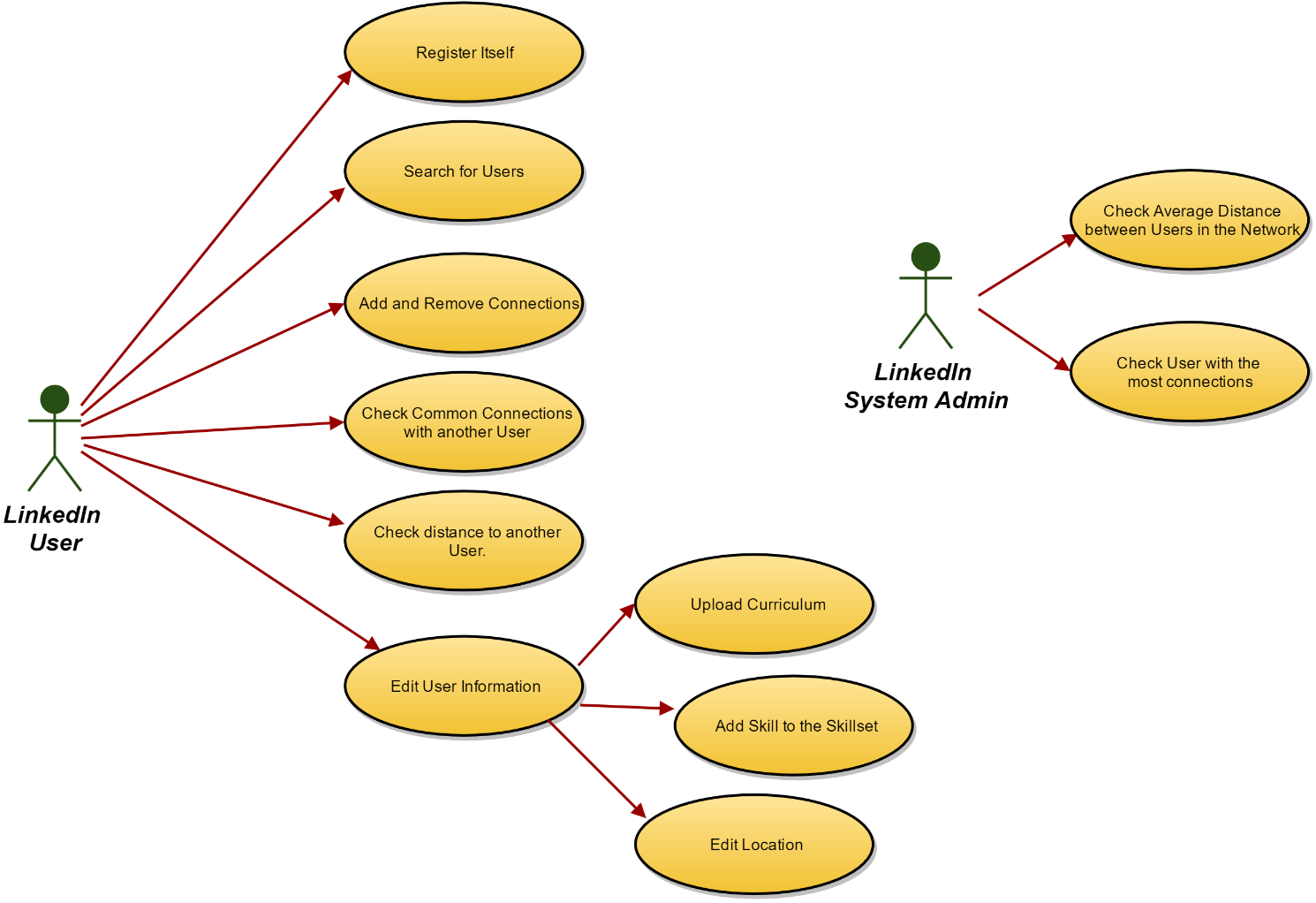
The actors in this system are the **LinkedIn Users**.

|  |  |  |
| --- | --- | --- |
| **Id** | **Priority** | **Description** |
| R1 | Mandatory | A user can register itself on the network, specifying their name. |
| R2 | Mandatory | A user can search for other users by their name. |
| R3 | Mandatory | A user has a list of connections and he can add or remove users to this list (one at a time). |
| R4 | Mandatory | A user can upload his CV information. |
| R5 | Mandatory | The system should be able to check the average distance between users in the network. |
| R6 | Mandatory | The system should be able to check the user with the most connections. |
| R7 | Mandatory | A user can query the system for the connections he has in common with another user. |
| R8 | Mandatory | The system should be able to check the distance between two users. |
| R9 | Mandatory | A user can add a skill to his list of skills. |
| R10 | Optional | A user can add and also edit his current location. |
|  |  |  |

These requirements are directly translated onto use cases as shown next.

# 2. Visual UML model

## 2.1 Use case model [[1]](#footnote-1)



The major use case scenarios (to be used later as test scenarios) are described next.

|  |  |
| --- | --- |
| **Scenario** | **Register** |
| **Description** | Normal scenario for a user to create his LinkedIn account. |
| **Pre-conditions** | 1. LinkedIn network has a non-null set of users. *(initial system state)*  2. The user provides a valid name, age and gender. *(input)* |
| **Post-conditions** | 1. A person instance is created with the provided info. *(final system state)*  2. The created person instance is added to the LinkedIn users. *(final system state)* |
| **Steps** | ---------------------------------------- |
| **Exceptions** | (unspecified) |

|  |  |
| --- | --- |
| **Scenario** | **Search for Users** |
| **Description** | Normal scenario for a User search. |
| **Pre-conditions** | 1. LinkedIn network has a non-null set of users. *(initial system state)*  2. The users all have a non-null name. *(initial system state)*  3. The name input for the search is a valid string. *(input)* |
| **Post-conditions** | 1. If a match is made, the set of users’ info (name, age, gender) are shown, otherwise the system emits a warning. *(output)* |
| **Steps** | ----------------------------------------- |
| **Exceptions** | 1. No match is made. |

|  |  |
| --- | --- |
| **Scenario** | **Add Connections** |
| **Description** | Scenario where the user adds a person (another user) as a connection. |
| **Pre-conditions** | 1. The users are registered in the network. *(initial system state)* |
| **Post-conditions** | 1. The added person is now part of the set of connections of the user. *(final system state)* |
| **Steps** | ------------------------------------------- |
| **Exceptions** | (unspecified) |

|  |  |
| --- | --- |
| **Scenario** | **Remove Connections** |
| **Description** | Scenario where the user removes a person (another user) as a connection. |
| **Pre-conditions** | 1. The users are registered in the network. *(initial system state)* |
| **Post-conditions** | 1. The connection between the users is removed in each of them. *(final system state)* |
| **Steps** | ------------------------------------------- |
| **Exceptions** | 1. There wasn’t a connection between the specified users |

|  |  |
| --- | --- |
| **Scenario** | **Check Common Connections with another User** |
| **Description** | Scenario where the user checks which people both he and another user are connected to. |
| **Pre-conditions** | 1. The user to check is registered in the network. *(initial system state)* |
| **Post-conditions** | 1. The number of common connections with the other user is bigger or equal to zero, which the system will display. *(output)* |
| **Steps** | ------------------------------------------- |
| **Exceptions** | (unspecified) |

|  |  |
| --- | --- |
| **Scenario** | **Check Distance to Another User** |
| **Description** | Normal scenario where a user checks how many connections away from him another user is. If a user has no connections, his distance is infinite. |
| **Pre-conditions** | 1. The users are registered in the network. *(initial system state)* |
| **Post-conditions** | 1. The minimum number of connections to reach the other user (distance) is bigger or equal to zero (in case it’s himself), which the system will display. *(output)* |
| **Steps** | ------------------------------------------- |
| **Exceptions** | 1. If either user has no connections – the distance to one another is infinite; |

|  |  |
| --- | --- |
| **Scenario** | **Upload Curriculum** |
| **Description** | Normal scenario in which a user uploads his whole CV into the system, taking the form of a “string” that represents the PDF or other type of file. |
| **Pre-conditions** | 1. The CV is a valid string type. *(input)* |
| **Post-conditions** | 1. The CV field is updated with the input. *(final system state)* |
| **Steps** | --------------------------------------------- |
| **Exceptions** | (unspecified) |

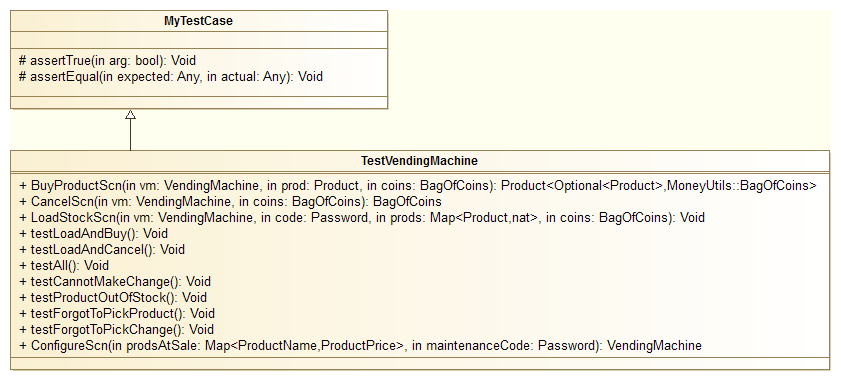
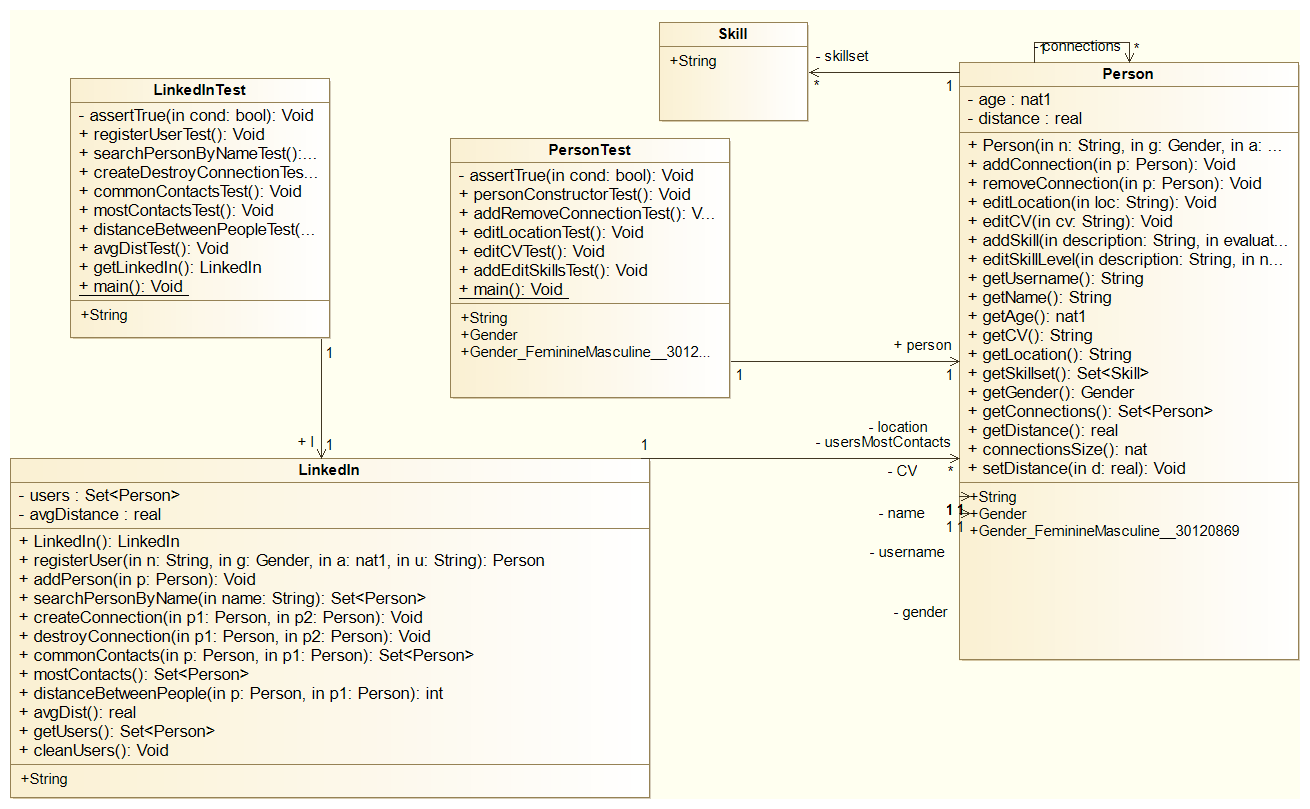
|  |  |
| --- | --- |
| **Scenario** | **Add Skill to the Skillset** |
| **Description** | Normal scenario in which a user adds a new skill to his set of existing skills. |
| **Pre-conditions** | 1. The skillset is a non-null set of skills, which it means it is initialized and has 0 or more items. *(initial system state)*  2. The skill to insert doesn’t exist in the user’s skillset yet. *(input)* |
| **Post-conditions** | 1. The skill is added to the user’s skillset. |
| **Steps** | ------------------------------------------------ |
| **Exceptions** | (unspecified) |

|  |  |
| --- | --- |
| **Scenario** | **Edit Location** |
| **Description** | Normal scenario in which a user edits the place of his current location. |
| **Pre-conditions** | 1. The new location is a valid string type. *(input)* |
| **Post-conditions** | 1. The user’s location is updated with the input. *(final system state)* |
| **Steps** | ---------------------------------------------------- |
| **Exceptions** | (unspecified) |

|  |  |
| --- | --- |
| **Scenario** | **Check Average Distance between Users in the network** |
| **Description** | Scenario where the system is requested to check the average distance between all the existing users. |
| **Pre-conditions** | 1. The network’s set of users isn’t null or empty. *(initial system state)* |
| **Post-conditions** | 1. The average distance between users is bigger or equal to zero (if there’s only 1 user), which the system will display. *(output)* |
| **Steps** | -------------------------------------------------- |
| **Exceptions** | 1. If no user has connections – the average distance is infinite;  2. If there are no users – the average distance is infinite; |

|  |  |
| --- | --- |
| **Scenario** | **Check User with the most connections** |
| **Description** | Scenario where the system is requested to check which existing user has the most connections. In case of draw, returns all the ones with the most connections. |
| **Pre-conditions** | 1. The network’s set of users isn’t null or empty. *(initial system state)* |
| **Post-conditions** | 1. The system displays the Person or set of People that have the most connections in the system. *(output)* |
| **Steps** | ------------------------------------------------- |
| **Exceptions** | 1. If there are no users, no match is made; |

## 2.2 Class model



|  |  |
| --- | --- |
| **Class** | **Description** |
| LinkedIn | Defines utility types and functions to work with bags (multisets) of coins.-- |
| Person | Defines a product at sale in a vending machine.-- |
| Skill | Skills a person has obtained and how proficient they are with said skill |
| PersonTest | Defines the test/usage scenarios and test cases for Person |
| LinkedInTest | Defines the test/usage scenarios and test cases for LinkedIn. |

# 3. Formal VDM++ model

## 3.1 Class LinkedIn

**class** LinkedIn

**types**

**public** String = **seq** **of** **char**;

**values**

**instance variables**

**private** users : **set** **of** Person := {};

**private** avgDistance : **real** := 1;

**private** usersMostContacts : **set** **of** Person := {};

**inv** card users >= 0;

**inv** avgDistance >= 0;

**operations**

**public** LinkedIn: () ==> LinkedIn

LinkedIn() == (

return self;

);

-- Register use case

**public** registerUser: String \* Person`Gender \* **nat1** \* String ==> Person

registerUser(n, g, a ,u) == (

**dcl** p:Person := new Person(n, g, a,u);

addPerson(p);

return p;

)

**pre** is\_String(n) and is\_Person`Gender(g) and is\_nat1(a) and is\_String(u) and users <> undefined

**post** (forall user **in set** users & (user <> RESULT => (user.getUsername() <> RESULT.getUsername())));

-- Auxiliar operation with auxiliar pre and post conds

**public** addPerson: Person ==> ()

addPerson(p) == users := users union {p}

**pre** p not **in set** users

**post** users = users~ union {p};

-- Search Person use case

**public** searchPersonByName : String ==> **set** **of** Person

searchPersonByName(name) ==

(

return {p | p **in set** users & p.getName() = name};

)

**pre** is\_String(name);-- exists p in set users & p.getName() = name -----> is this part needed?

-- Add Connection use case

**public** createConnection: Person\*Person ==> ()

createConnection(p1, p2) == (

p1.addConnection(p2);

p2.addConnection(p1);

)

**pre** p1 in **set** users and p2 in **set** users and p1<>p2

**post** p2 in **set** p1.getConnections() and p1 in **set** p2.getConnections();

-- Destroy Connection use case

**public** destroyConnection: Person\*Person ==> ()

destroyConnection(p1, p2) == (

p1.removeConnection(p2);

p2.removeConnection(p1);

)

**pre** p1<>p2 and p1 in **set** users and p2 in **set** users and p2 in **set** p1.getConnections() and p1 in **set** p2.getConnections()

**post** p2 not **in set** p1.getConnections() and p1 not **in set** p2.getConnections();

-- Common Contacts use case

**public** commonContacts : Person\*Person ==> **set** **of** Person

commonContacts(p,p1) == (

-- dcl commonContacts: set of Person :-= {};

--for all c in set p.getConnections() do if (c in set p1.getConnections())s

-- then commonContacts := commonContacts union {c};

--return commonContacts;

return {user | user **in set** p1.getConnections() & user in **set** p.getConnections() }

)

**pre** p <> p1 and exists p2,p3 **in set** users & ( p2=p and p3 =p1)

**post** not exists u **in set** p1.getConnections() & (u in **set** p.getConnections() and u not **in set** RESULT);

--exists p2,p3 in set users & ( p2 = p and p3 = p1 and

--(

-- forall c in set RESULT & (c in set p2.getConnections() and c in set p3.getConnections()

-- )));

-- Most Contacts use case

**public** mostContacts:() ==> **set** **of** Person

mostContacts() ==

return {s|s **in set** users & (not exists p1 **in set** users & p1.connectionsSize() > s.connectionsSize())}

**pre** users <> {}

**post** forall u **in set** RESULT & (not exists p **in set** users & (p.connectionsSize() > u.connectionsSize()));

-- Distance Between Two Users use case

**public** distanceBetweenPeople:Person\*Person ==> **int**

distanceBetweenPeople(p,p1) ==(

**dcl** queue : **seq** **of** Person := [];**dcl** current : Person ;

for **all** u **in set** users **do** (

u.setDistance(-1);

);

p.setDistance(0);

queue := queue ^ [p] ;

while queue <> [] **do** (

current := hd queue;

queue := tl queue;

for **all** c **in set** current.getConnections() **do** (

if(c.getDistance() = -1) **then** (

c.setDistance( current.getDistance()+1);

queue:= queue ^ [c] ;

);

if(c = p1) **then**

return c.getDistance();

);

);

-- no path connecting the two people. Returns -1

return -1;

)

**pre** users <> {} and exists p2,p3 **in set** users & (p=p2 and p1=p3);

--post RESULT = -1 or (p = p1 and RESULT = 0);

-- Average Distance use case

**public** avgDist: () ==> **real**

avgDist() == (

**dcl** avg: **int** := 0;

**dcl** cnt: **int** := 0;

for **all** u **in set** users **do** (

**dcl** users2: **set** **of** Person := users \ {u};

for **all** uu **in set** users2 **do** (

**dcl** dst: **int** := distanceBetweenPeople(u, uu);

if(dst <> -1) **then** (

avg := avg + dst;

cnt := cnt + 1;

)

);

);

if(cnt = 0)

**then**

return 0

**else**

avgDistance := avg/cnt;

return avgDistance;

)

**pre** users <> {} and users <> undefined

**post** avgDistance >= 0;-- should I leave this here?

-- Auxiliar functions

**public** **pure** getUsers : () ==> **set** **of** Person

getUsers() == return users;

**public** cleanUsers : () ==> ()

cleanUsers() ==( users :={};

return)

**post** users = {};

**end** LinkedIn

## 3.2 Class Person

**class** Person

**types**

**public** String = **seq** **of** **char**;

**public** Gender = **<Masculine>** | **<Feminine>**;

**values**

-- TODO Define values here

**instance variables**

**private** connections : **set** **of** Person := {};

**private** name : String :="";

**private** age : **nat1** := 99;

**private** gender: Gender := <Masculine>;

**private** CV: String :="";

**private** skillset : **set** **of** Skill := {};

**private** location : String := "";

**private** distance : **real** := 0;

**private** username : String;

--it worked devolveu o caminho mais curto, só tive que alterar duas linhas

**operations**

**public** Person: String \* Gender \* **nat1** \* String ==> Person

Person(n, g, a,u) == (

name:=n; gender:=g; age:=a;username:=u;

return self;

) **pre** is\_String(n) and is\_Person`Gender(g) and is\_nat1(a) and is\_String(u);

**public** addConnection: Person ==> ()

addConnection(p) == (connections := connections union {p};

)

**pre** p not **in set** connections

**post** connections = connections~ union {p};

**public** removeConnection: Person ==> ()

removeConnection(p) == connections := connections\{p}

**pre** p in **set** connections

**post** connections = connections~\{p};

**public** editLocation: String ==> ()

editLocation(loc) == location := loc

**pre** is\_String(loc);

**public** editCV: String ==> ()

editCV(cv) == CV:=cv

**pre** is\_String(cv);

**public** addSkill: String \* **nat1** ==> ()

addSkill(description, evaluation) == (

**dcl** skill:Skill := new Skill(description, evaluation);

skillset:= skillset union {skill}

)

**pre** skillset <> undefined and (not exists s **in set** skillset & (s.getDescription() = description))

**post** exists s **in set** skillset & (s.getDescription() = description and s.getLevel() = evaluation

and skillset = skillset~ union {s});

-- Assume a person never completely loses a skill, only forgets it a bit over time, so evaluation may be edited

**public** editSkillLevel: String\***nat1** ==> ()

editSkillLevel(description, newLevel) == (

for **all** skills **in set** skillset **do**(

if(skills.getDescription() = description) **then**

skills.setLevel(newLevel)

)

)**pre** skillset <> undefined and (card skillset >0) and exists s **in set** skillset & (s.getDescription() = description)

**post** exists s **in set** skillset & (s.getDescription() = description and s.getLevel() = newLevel );

**public** **pure** getUsername : () ==> String

getUsername() == return username;

**public** **pure** getName : () ==> String

getName() == return name;

**public** **pure** getAge : () ==> **nat1**

getAge() == return age;

**public** **pure** getCV : () ==> String

getCV() == return CV;

**public** **pure** getLocation : () ==> String

getLocation() == return location;

**public** **pure** getSkillset : () ==> **set** **of** Skill

getSkillset() == return skillset;

**public** **pure** getGender : () ==> Gender

getGender() == return gender;

**public** **pure** getConnections : () ==> **set** **of** Person

getConnections() == return connections;

**public** **pure** getDistance : () ==> **real**

getDistance() == return distance;

**public** **pure** connectionsSize: () ==> **nat**

connectionsSize() == return card connections;

**public** setDistance : **real** ==> ()

setDistance(d) == distance := d;

**functions**

**traces**

-- TODO Define Combinatorial Test Traces here

## end Person

## 3.3 Class Skill

**class** Skill

**types**

**public** String = **seq** **of** **char**;

**values**

-- TODO Define values here

**instance variables**

**private** description : String := "";

**private** level : **nat1** := 1;

**operations**

**public** Skill: String \* **nat1** ==> Skill

Skill(desc, lev) == (

description := desc; level := lev;

return self;

);

**public** **pure** getDescription : () ==> String

getDescription() == return description;

**public** setDescription : String ==> ()

setDescription(desc) == description:= desc;

**public** **pure** getLevel : () ==> **nat1**

getLevel() == return level;

**public** setLevel : **nat1** ==> ()

setLevel(lev) == level:=lev;

**functions**

-- TODO Define functiones here

**traces**

-- TODO Define Combinatorial Test Traces here

# end Skill

# 4. Model validation

## 4.1 Class LinkedInTest

**class** LinkedInTest

**types**

**public** String = **seq** **of** **char**;

**values**

**instance variables**

**public** l : LinkedIn := new LinkedIn();

**operations**

assertTrue: **bool** ==> ()

assertTrue(cond) == return

**pre** cond;

-- Register users

**public** registerUserTest: () ==> ()

registerUserTest() ==

(

**dcl** p1:Person , p2:Person, p3:Person;

assertTrue(card l.getUsers() = 0);

p1 := l.registerUser("Luis",**<Masculine>**,22,"luis22");

assertTrue(card l.getUsers() = 1);

assertTrue(forall s **in set** l.getUsers() & s.getName() = "Luis"); -- Only ppl named luis exist yet

assertTrue(exists1 s **in set** l.getUsers() & s.getName() = "Luis"); -- Only ppl named luis exist yet

p2 := l.registerUser("Luis",**<Masculine>**,23,"luis23");

assertTrue(card l.getUsers() = 2);

assertTrue(forall s **in set** l.getUsers() & s.getName() = "Luis"); -- Only ppl named luis exist yet

p3 := l.registerUser("Carla",**<Feminine>**,30,"carla30");

assertTrue(card l.getUsers() = 3);

assertTrue(exists1 s **in set** l.getUsers() & s.getName() = "Carla"); -- Now we have 2 Luis and one Carla

-- NO NEED TO TEST IF PERSON CONSTRUCTORS SET PARAMETERS CORRECTLY - this is already done on the tests of the PersonTest class, specifically personConstructorTest

)

**pre** l.getUsers() = {} and card l.getUsers() = 0

**post** l.getUsers() <> {} and card l.getUsers() = 3;

-- Search Person By Name

**public** searchPersonByNameTest: () ==> ()

searchPersonByNameTest() == (

**dcl** p1:Person, p2:Person;

**dcl** pplSearched:**set** **of** Person;

assertTrue(0=card l.getUsers());

p1 := l.registerUser("Luis",**<Masculine>**,22,"luis22");

p2 := l.registerUser("Luis",**<Masculine>**,23,"luis23");

-- Search for Luis -> Will find 2 people, both really named "Luis"

pplSearched := l.searchPersonByName("Luis");

assertTrue(card pplSearched=2);

for **all** p **in set** pplSearched **do**

assertTrue(p.getName()="Luis");

-- Search for Joao -> Will find no one with that name

pplSearched := l.searchPersonByName("Joao");

assertTrue(card pplSearched = 0);

return

);

-- Add and remove connections

**public** createDestroyConnectionTest: () ==> ()

createDestroyConnectionTest() == (

**dcl** p:Person;

**dcl** p1:Person;

**dcl** p2: Person;

p:=l.registerUser("Luis",**<Masculine>**,22,"luis22");

p1:=l.registerUser("Filipe",**<Masculine>**,22,"filipe22");

p2:=l.registerUser("Maria",**<Feminine>**, 24, "maria24");

l.createConnection(p,p1);

l.createConnection(p1,p2);

assertTrue(card p.getConnections() = 1 and card p1.getConnections() = 2 and card p2.getConnections() = 1);

-- Luis has 1 connection with username filipe22

assertTrue(exists s **in set** p.getConnections() & s.getUsername()="filipe22");

-- Filipe has 2 connections with usernames "luis22" and "maria24"

assertTrue(exists1 s **in set** p1.getConnections() & s.getUsername()="luis22");

assertTrue(exists1 s **in set** p1.getConnections() & s.getUsername()="maria24");

--Maria has 1 connection with username "filipe22"

assertTrue(exists1 s **in set** p2.getConnections() & s.getUsername()="filipe22");

-- P and P1 stop being connected

l.destroyConnection(p,p1);

assertTrue(card p.getConnections() = 0 and card p1.getConnections() = 1 and card p2.getConnections() = 1);

-- Filipe now only has 1 connection, to "maria24" and she only has 1 connection, to "filipe22"

assertTrue(exists1 s **in set** p1.getConnections() & s.getUsername()="maria24");

assertTrue(exists1 s **in set** p2.getConnections() & s.getUsername()="filipe22");

l.destroyConnection(p1,p2);

assertTrue(card p.getConnections() = 0 and card p1.getConnections() = 0 and card p2.getConnections() = 0);

return

);

-- Common Contacts

**public** commonContactsTest: () ==> ()

commonContactsTest() ==

(

**dcl** p:Person;

**dcl** p1:Person;

**dcl** p2:Person;

**dcl** p3:Person;

assertTrue(0=card l.getUsers());

p:=l.registerUser("Luis",**<Masculine>**,22,"luis22");

p1:=l.registerUser("Filipe",**<Masculine>**,22,"filipe22");

p2:=l.registerUser("Pedro",**<Masculine>**,22,"pedro22");

p3:=l.registerUser("Joana",**<Feminine>**,22,"joana22");

l.createConnection(p1,p2);

l.createConnection(p,p2);

l.createConnection(p2,p3);

-- p1 and p1 have p2 as common

assertTrue(l.commonContacts(p1,p) = {p2});

-- p2 and p3 have no common

assertTrue(l.commonContacts(p2,p3) = {});

);

-- User with the most contacts

**public** mostContactsTest: () ==> ()

mostContactsTest() == (

**dcl** p:Person;

**dcl** p1:Person;

**dcl** p2:Person;

**dcl** p3:Person;

assertTrue(0=card l.getUsers()); -- this is already tested on registerUserTest

p:=l.registerUser("Luis",**<Masculine>**,22,"luis22");

p1:=l.registerUser("Filipe",**<Masculine>**,22,"filipe22");

p2:=l.registerUser("Pedro",**<Masculine>**,22,"pedro22");

p3:=l.registerUser("Maria",**<Feminine>**, 24,"maria24");

-- As long as there are users, there's always someone with the most, even if that most is 0.

-- Before any connections, they all have 0, so they all have the most connections.

assertTrue({p,p1,p2,p3} = l.mostContacts());

-- p<->p1<->p2 Therefore p1 has 1 connections, p1 has 2 and p2 has 1. p1 has the most connections!

l.createConnection(p,p1);

l.createConnection(p1,p2);

assertTrue(l.mostContacts() = {p1});

-- p<->p1<->p2<->p3 Therefore p has 1 connections, p1 has 2, p2 has 2 and p3 has 1. p1 and p2 are tied for the most connections!

l.createConnection(p2, p3);

assertTrue(l.mostContacts() = {p1,p2});

-- <-/ /-> Means a broken link

-- p<-/ /->p1<->p2<->p3 Therefore p has 0 connections, p1 has 1, p2 has 2 and p3 has 1. p2 has the most connections!

l.destroyConnection(p,p1);

assertTrue(l.mostContacts() = {p2});

-- ...p<->p1<->p2<->p3<->p... it's a full ring cycle. Every person has 2 connections.

l.createConnection(p, p1);

l.createConnection(p3, p);

assertTrue(l.mostContacts() = {p,p1,p2,p3});

);

-- Distance Between People

**public** distanceBetweenPeopleTest: () ==> ()

distanceBetweenPeopleTest() == (

**dcl** p:Person;

**dcl** p1:Person;

**dcl** p2:Person;

**dcl** p3:Person;

assertTrue(card l.getUsers() = 0);

p:=l.registerUser("Luis",**<Masculine>**,22,"luis22");

p1:=l.registerUser("Filipe",**<Masculine>**,22,"filipe22");

p2:=l.registerUser("Pedro",**<Masculine>**,22,"pedro22");

p3:=l.registerUser("Jose",**<Masculine>**,22,"jose22");

assertTrue(card l.getUsers() = 4);

-- p<->p1<->p2 p3

l.createConnection(p,p1);

l.createConnection(p1,p2);

-- d(p,p1)=1; d(p1,p2)=1; d(p,p2)=2; d(p,p)=0 as it is itself

-- d(p, p3)=-1, as there is no path between them, same for p1 to p3 and p2 to p3

assertTrue(l.distanceBetweenPeople(p,p1) = 1);

assertTrue(l.distanceBetweenPeople(p,p2) = 2);

assertTrue(l.distanceBetweenPeople(p,p) = 0);

assertTrue(l.distanceBetweenPeople(p,p3) = -1);

assertTrue(l.distanceBetweenPeople(p1,p3) = -1);

assertTrue(l.distanceBetweenPeople(p2,p3) = -1);

return

);

-- TODO put 1 more case with a disconnected graph

-- Average Distance between users in the network

**public** avgDistTest : () ==> ()

avgDistTest() == (

**dcl** p:Person:= l.registerUser("Pedro",**<Masculine>**,22,"pedro22");

**dcl** p1:Person:= l.registerUser("Luis",**<Masculine>**,22,"luis22");

**dcl** p2:Person:= l.registerUser("Carlos",**<Masculine>**,22,"carlos22");

**dcl** p3:Person:= l.registerUser("Filipe",**<Masculine>**,22,"filipe22");

**dcl** p4:Person;

assertTrue(0 = l.avgDist());

l.createConnection(p,p1);

l.createConnection(p3,p1);

l.createConnection(p2,p3);

l.createConnection(p2,p);

/\*

p - p1 = 1

p - p2 = 1

p - p3 = 2

p1 - p2 = 2

p1 - p3 = 1

p2 - p3 = 1

distance:= 1+1+2+2+1+1= 8

nConnections := 6;

avgDist := 8/6

\*/

assertTrue(8/6 = l.avgDist());

p4:= l.registerUser("Jose",**<Masculine>**,22,"jose22");

--Stays the same since P4 is not connected to any Person in the network thefore it has no connections in the network

assertTrue(8/6 = l.avgDist());

);

**public** **pure** getLinkedIn:() ==> LinkedIn

getLinkedIn() ==

return l;

-- MAIN

**public** **static** main: () ==> ()

main() == (

**dcl** lt: LinkedInTest :=new LinkedInTest();

lt.registerUserTest();

lt.getLinkedIn().cleanUsers();

lt.searchPersonByNameTest();

lt.getLinkedIn().cleanUsers();

lt.mostContactsTest();

lt.getLinkedIn().cleanUsers();

lt.commonContactsTest();

lt.getLinkedIn().cleanUsers();

lt.avgDistTest();

lt.getLinkedIn().cleanUsers();

lt.createDestroyConnectionTest();

lt.getLinkedIn().cleanUsers();

lt.distanceBetweenPeopleTest();

PersonTest`main();

return

);

**functions**

**traces**

**end** LinkedInTest

## 4.2 Class PersonTest

**class** PersonTest

**types**

**public** String = **seq** **of** **char**;

**public** Gender = **<Masculine>** | **<Feminine>**;

**values**

-- TODO Define values here

**instance variables**

**public** person:Person := new Person("Teste",<Masculine>,22,"Teste22");

**operations**

-- Assert Generic Operation

assertTrue: **bool** ==> ()

assertTrue(cond) == return

**pre** cond;

-- Test Person constructor (Name, gender, age, username and default values not input)

**public** personConstructorTest: () ==> ()

personConstructorTest() == (

**dcl** testPerson:Person := new Person("Teste",<Masculine>,22,"Teste22");

assertTrue(testPerson.getName() = "Teste");

assertTrue(testPerson.getGender() = <Masculine>);

assertTrue(testPerson.getAge() = 22);

assertTrue(testPerson.getUsername() = "Teste22");

assertTrue(testPerson.getCV() = "");

assertTrue(testPerson.getLocation() = "");

assertTrue(card testPerson.getSkillset() = 0);

assertTrue(card testPerson.getConnections() = 0);

return

);

-- Add and remove connections only on this user's side (doesn't affect the symmetrical connection on the user he is adding or removing)

**public** addRemoveConnectionTest: () ==> ()

addRemoveConnectionTest() == (

-- Test that person starts out with 0 connections

**dcl** p1:Person := new Person("Pedro",<Masculine>,22,"Pedro22");

**dcl** p2:Person := new Person("Joana", <Feminine>, 45, "Joana45");

**dcl** p3:Person := new Person("Ana", <Feminine>, 19, "Ana19");

assertTrue(card p1.getConnections() = 0);

assertTrue(card p1.getConnections() = 0);

-- Pedro adds Joana to his connection set

p1.addConnection(p2);

assertTrue(card p1.getConnections() = 1);

assertTrue(forall p **in set** p1.getConnections() & p.getName() = "Joana");

-- Pedro adds Ana to his connection set

p1.addConnection(p3);

assertTrue(card p1.getConnections() = 2);

assertTrue(exists1 p **in set** p1.getConnections() & p.getName() = "Ana");

-- Pedro removes Joana from his connection set

p1.removeConnection(p2);

assertTrue(card p1.getConnections() = 1);

assertTrue(forall p **in set** p1.getConnections() & p.getName() = "Ana");

-- Pedro removes Ana from his connection set

p1.removeConnection(p3);

assertTrue(card p1.getConnections() = 0);

return

);

-- Edit User Location

**public** editLocationTest: () ==> ()

editLocationTest() == (

person.editLocation("Location String");

assertTrue(person.getLocation() = "Location String");

assertTrue(len person.getLocation() = 15);

person.editLocation("");

assertTrue(person.getLocation() = "");

assertTrue(len person.getLocation() = 0);

return

);

-- Edit User Curriculum

**public** editCVTest: () ==> ()

editCVTest() == (

person.editCV("TestCV String");

assertTrue(person.getCV() = "TestCV String");

assertTrue(len person.getCV() = 13);

person.editCV("");

assertTrue(person.getCV() = "");

assertTrue(len person.getCV() = 0);

return

)

**pre** is\_String(person.getCV());

-- Add skills and edit their evaluations

**public** addEditSkillsTest: () ==> ()

addEditSkillsTest() == (

**dcl** p1:Person := new Person("Pedro",<Masculine>,22,"Pedro22");

assertTrue(card p1.getSkillset() = 0);

-- Add a first skill

p1.addSkill("C++", 4);

assertTrue(card p1.getSkillset() = 1);

assertTrue(exists1 p **in set** p1.getSkillset() & p.getDescription() = "C++" and p.getLevel() = 4);

-- Add a second skill

p1.addSkill("Java", 5);

assertTrue(card p1.getSkillset() = 2);

assertTrue(exists1 p **in set** p1.getSkillset() & p.getDescription() = "Java" and p.getLevel() = 5);

-- Edits a skill evaluation, forgot a bit of C++

p1.editSkillLevel("C++", 3);

assertTrue(card p1.getSkillset() = 2);

assertTrue(exists1 p **in set** p1.getSkillset() & p.getDescription() = "C++" and p.getLevel() = 3);

return

);

**public** **static** main: () ==> ()

main() ==

(

**dcl** personTest: PersonTest :=new PersonTest();

personTest.personConstructorTest();

personTest.addRemoveConnectionTest();

personTest.editLocationTest();

personTest.editCVTest();

personTest.addEditSkillsTest();

return

);

# end PersonTest

# 5. Model verification

## 5.1 Example of domain verification

One of the proof obligations generated by Overture is:

|  |  |  |
| --- | --- | --- |
| No. | PO Name | Type |
| 61 | VendingMachine`setProductsAtSale | legal map application |

The code under analysis (with the relevant map application underlined) is:

**public** setProductsAtSale: **map** ProductName **to** ProductPrice ==> ()

setProductsAtSale(prods) ==

products := { name |-> **new** Product(name, prods(name)) | **name in set dom prods**};

In this case the proof is trivial because the quantification '**name in set dom prods'** assures that the map is accesses only inside its domain.

## 5.2 Example of invariant verification

Another proof obligation generated by Overture is:

|  |  |  |
| --- | --- | --- |
| No. | PO Name | Type |
| 73 | VendingMachine`pickChange | state invariant holds |

The code under analysis (with the relevant state changes underlined) is:

**public** pickChange: () ==> BagOfCoins

pickChange() ==

**let** r = changeToPickup **in** (

**atomic**(

changeToPickup := {|->};

status := **if** productToPickup = **nil** **then** **<Idle>** **else** **<Delivering>**

);

**return** r

)

**pre** changeToPickup **<>** {|->};

The relevant invariant under analysis is:

**inv** changeToPickup **<>** {|->} **or** productToPickup **<>** **nil** <=> status = **<Delivering>**;

After the execution of the '**atomic**' block we have (technically, this is the post-condition of the block):

changeToPickup = {|->} **and** (status = **if** productToPickup = **nil** **then** **<Idle>** **else** **<Delivering>)**

We have to prove that this implies that the invariant holds, i.e., that the following condition holds:

(changeToPickup = {|->} **and** (status = **if** productToPickup = **nil** **then** **<Idle>** **else** **<Delivering>)) =>**

**(**changeToPickup **<>** {|->} **or** productToPickup **<>** **nil** <=> status = **<Delivering>)**

This formally implies that:

(status = **if** productToPickup = **nil** **then** **<Idle>** **else** **<Delivering>)**

**=> (**productToPickup **<>** **nil** <=> status = **<Delivering>)**

which can be rewritten as:

(**if** productToPickup = **nil** **then** status = **<Idle>** **else** status = **<Delivering>)**

**=> (if** productToPickup **=** **nil** **then** status <> **<Delivering>) else** status = **<Delivering>)**

which is obviously true.

# 6. Conclusions

The model that was developed covers all the requirements.

If space permitted, as future work, it would be useful to also specify and test the behavior in exceptional conditions.

This project took approximately 16 hours to develop.

# 7. References

1. Validated Designs for Object-oriented Systems, J. Fitzgerald, P.G. Larsen, P. Mukherjee, N. Plat, M. Verhoef, Springer, 2005
2. VDM-10 Language Manual, Peter Gorm Larsen et al, Overture Technical Report Series No. TR-001, March 2014
3. Overture tool web site, http://overturetool.org

1. **Tip**: Some use cases are more general and get divided into their sub-components, like “Edit User Information”, which is divided into the various ways an user can edit his info. [↑](#footnote-ref-1)