“Railway Signaling System”

Relatório Final

4º Ano do Mestrado Integrado em Engenharia Informática e Computação

Métodos Formais em Engenharia de Software

Elementos do grupo:

João Pedro Miranda Maia | 201206047 | ei12089@fe.up.pt

Miguel Oliveira Sandim | 201201770 | ei12061@fe.up.pt

14 de dezembro de 2015

1 - Descrição Informal do Sistema

O modelo em análise neste relatório corresponde a um sistema de sinalização para uma linha ferroviária única circular que se bifurca ao chegar a uma estação. O objectivo deste sistema é o de gerir o tráfego ferroviário e impedir que exista a colisão entre comboios.

Assim, considera-se que a linha, entre estações, se encontra dividida em blocos. Antes de cada bloco, existe um semáforo que indica se a entrada no próximo bloco é permitida (cor verde ou amarela) ou impedida (cor vermelha), sendo apenas permitida a permanência de um comboio em cada bloco. Associado a cada semáforo existe também um sensor que detecta a entrada de comboios no bloco e altera o semáforo conforme. A regra geral para a sinalização indica que caso um bloco esteja ocupado por um comboio, o semáforo imediatamente antes desse bloco estará vermelho, e o anterior amarelo.

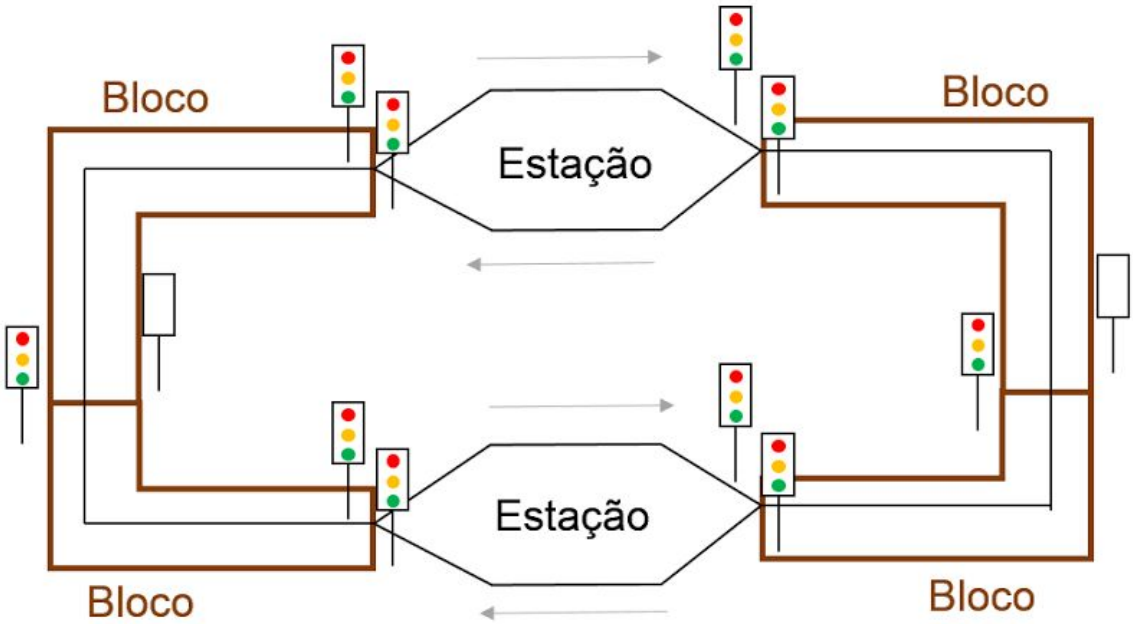

Fig. 1 - Esquema que representa uma instância de uma linha ferroviária a modelar com duas estações e dois blocos entre estação.

Em cada estação a linha é bifurcada para possibilitar a passagem de dois comboios em simultâneo, pelo que existem dois blocos por estação (que apresentam as mesmas regras descritas anteriormente). Os comboios neste caso circulam sempre pela linha à sua esquerda. O semáforo imediatamente depois de uma estação encontra-se sempre vermelho (o seu estado só muda para verde/amarelo quando um comboio numa estação pressiona um botão para pedir para sair dessa estação e avançar). Assim, o semáforo que gere as entradas numa estação estará sempre amarelo ou vermelho.

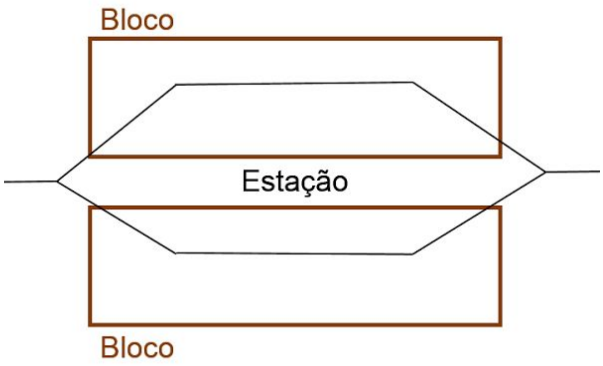

Fig. 2 - Esquema que representa a representação de uma estação em dois blocos.

1.1 - Implementação do Modelo

1.1.1 - Implementação do sentido de orientação

Relativamente à implementação do modelo, definiu-se que o “circular pela esquerda” ou “circular pela direita” é, respectivamente, traduzido por uma orientação do comboio “UP” ou “DOWN” [1], tal como exemplifica a *Figura 3*.

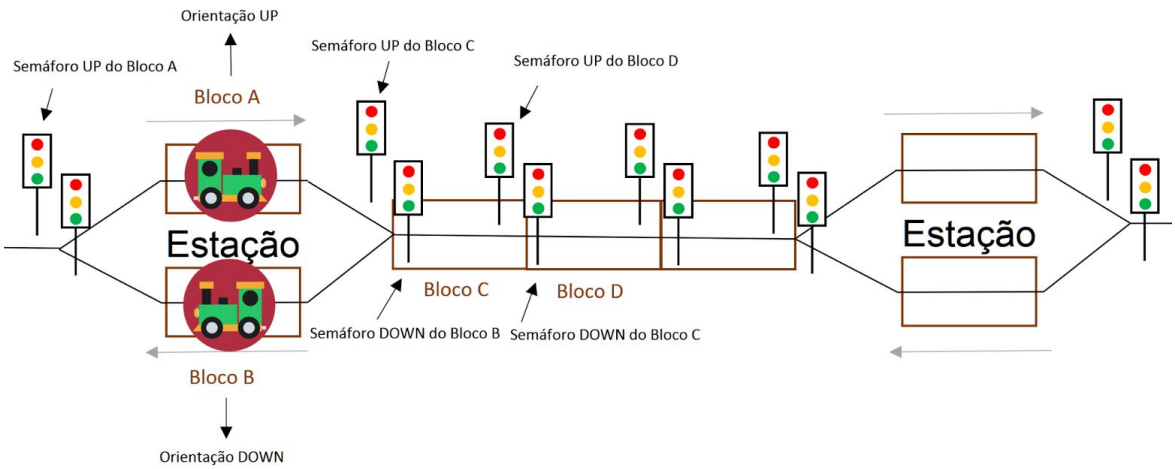

Fig. 3 - Esquema que exemplifica a nomenclatura dada à orientação dos comboios (“UP”/”DOWN”) e aos semáforos.

1.1.2 - Implementação da existência de travagens de emergência

Para evitar que comboios se acumulem entre estações devido ao facto de um comboio ter feito uma travagem de emergência, definiu-se que um pedido de saída de uma estação apenas deve ser aprovado pela estação central, caso não exista nenhum sensor ocupado (ou indisponível) durante o caminho até á próxima estação (inclusive) (ver *Figura 4*). Desta forma, assegura-se que caso exista uma travagem inesperada, os comboios estão sempre parados numa estação, e garante que caso um comboio parta de uma estação não possui nenhum obstáculo até entrar no bloco da esquerda da próxima estação.

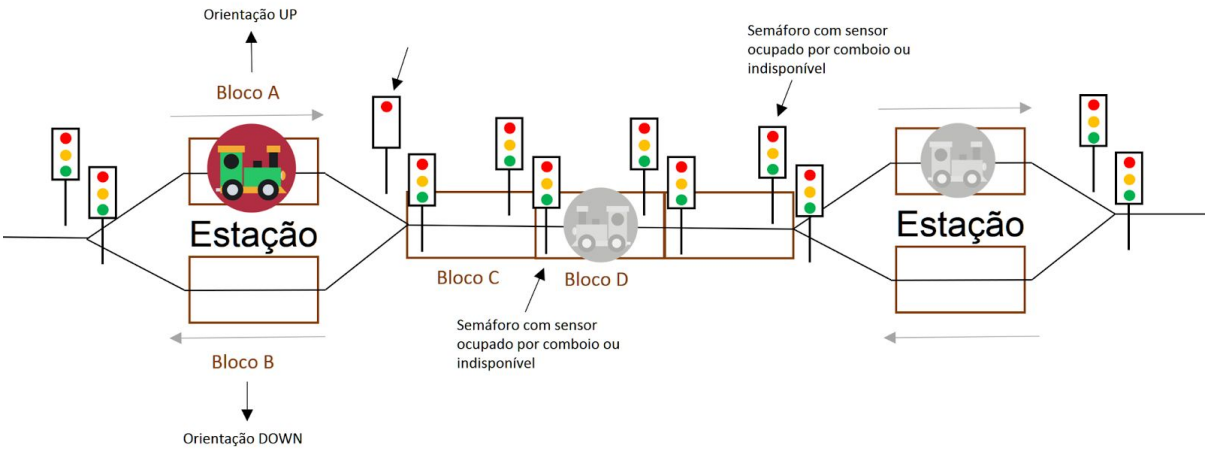

Fig. 4 - Esquema que explica as situações que levam à rejeição de um pedido de saída da estação por parte do comboio no Bloco A.

1.1.3 - Implementação da existência de semáforos/sensores indisponíveis durante a movimentação do comboio

Para tratar o caso em que um comboio, durante a viagem até uma próxima estação, encontra um semáforo indisponível (ou um sensor indisponível, que resultará num semáforo indisponível também, já que os sensores controlam os semáforos respectivos), quando um comboio encontra um semáforo indisponível à sua frente não se pode movimentar até que este fique verde ou amarelo (ver *Figura 5*).

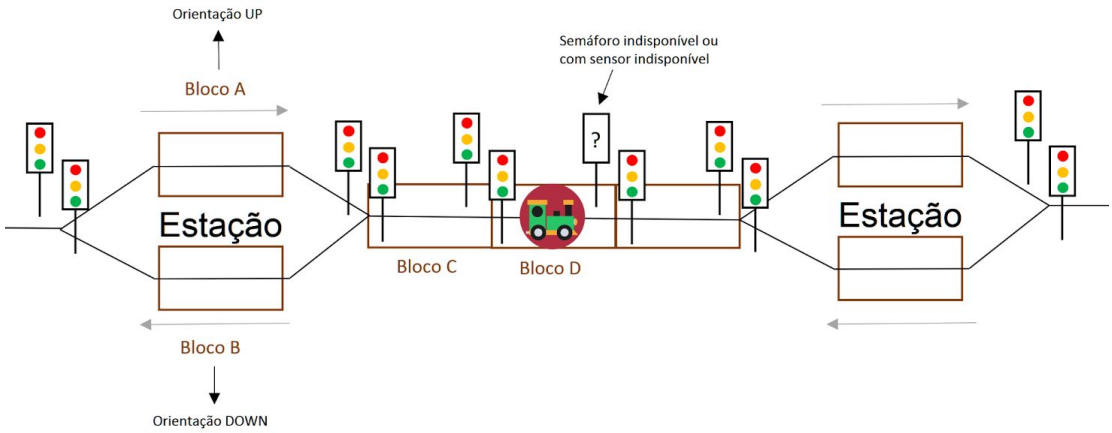

Fig. 5 - Esquema que explica a situação em que um comboio não se pode mover porque um semáforo/sensor está indisponível.

1.2 - Lista de Requisitos

Apresenta-se de seguida a lista de requisitos do sistema:

ID	Prioridade	Descrição
R1	**Obrigatório**	A linha férrea é circular.
R2	**Obrigatório**	A linha é constituída por uma faixa entre estações e duas faixas

		nas estações.
R3	**Obrigatório**	A linha férrea tem de ser constituída por, pelo menos, duas estações.
R4	**Obrigatório**	Tem de existir, pelo menos, um bloco entre duas estações.
R5	**Obrigatório**	A linha férrea é representada por uma sequência de blocos, excepto nas estações em que é constituída por dois blocos paralelos.
R6	**Obrigatório**	No início de cada bloco existe um sensor para detectar a entrada de um comboio, que pode estar indisponível.
R7	**Obrigatório**	Antes de cada bloco, existe um semáforo a indicar se um comboio no bloco anterior pode entrar no bloco seguinte, podendo estar vermelho, amarelo, verde ou indisponível.
R8	**Obrigatório**	Os comboios devem sempre parar quando chegarem a um bloco onde existe uma estação.
R9	**Obrigatório**	Nas estações, os comboios circulam pela esquerda.
R10	**Obrigatório**	Não pode haver mais do que um comboio a circular num bloco ao mesmo tempo.
R11	**Obrigatório**	Um comboio não pode sair de uma estação se um dos sensores até à próxima estação estiver ocupado ou avariado, ou se o sensor da próxima estação estiver ocupado ou avariado.
R12	**Obrigatório**	Quando um comboio entra num bloco, o semáforo anterior fica vermelho, e o semáforo antes desse fica amarelo (ou vermelho, caso seja o semáforo de saída de uma estação).
R13	**Obrigatório**	Quando um comboio entra num bloco, o penúltimo semáforo anterior fica verde (excepto quando é o de entrada numa estação, ficando amarelo, ou quando é o de saída de uma estação, ficando vermelho).
R14	**Obrigatório**	O maquinista de um comboio deve poder avançar o comboio de um bloco para o seguinte.
R15	**Obrigatório**	O maquinista de um comboio deve poder parar o comboio numa estação ou num bloco entre estações caso seja uma paragem de emergência.
R16	**Obrigatório**	O maquinista de um comboio deve poder pedir permissão ao centro de controlo para poder abandonar uma estação, esperando que o semáforo para avançar fique verde ou amarelo.
R17	**Obrigatório**	O operador da central de controlo deve poder ligar e desligar semáforos da linha ferroviária.

R18	**Obrigatório**	O operador da central de controlo deve poder ligar e desligar sensores da linha ferroviária.
R19	**Obrigatório**	O sensor associado a um semáforo deve poder alterar o estado desse semáforo (cor) quando um comboio passa por cima deste.

2 - Modelagem UML

2.1 - Casos de Utilização

Apresenta-se de seguida o Diagrama de Casos de Uso do modelo implementado:

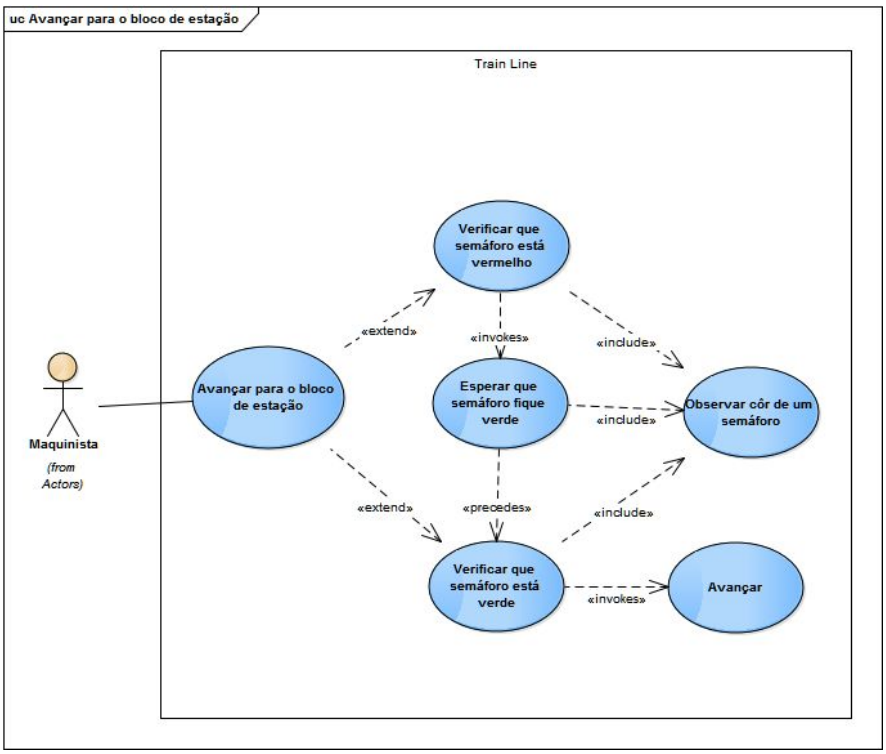

Fig. 6 - Caso de utilização correspondente ao avançar para um bloco de uma estação.

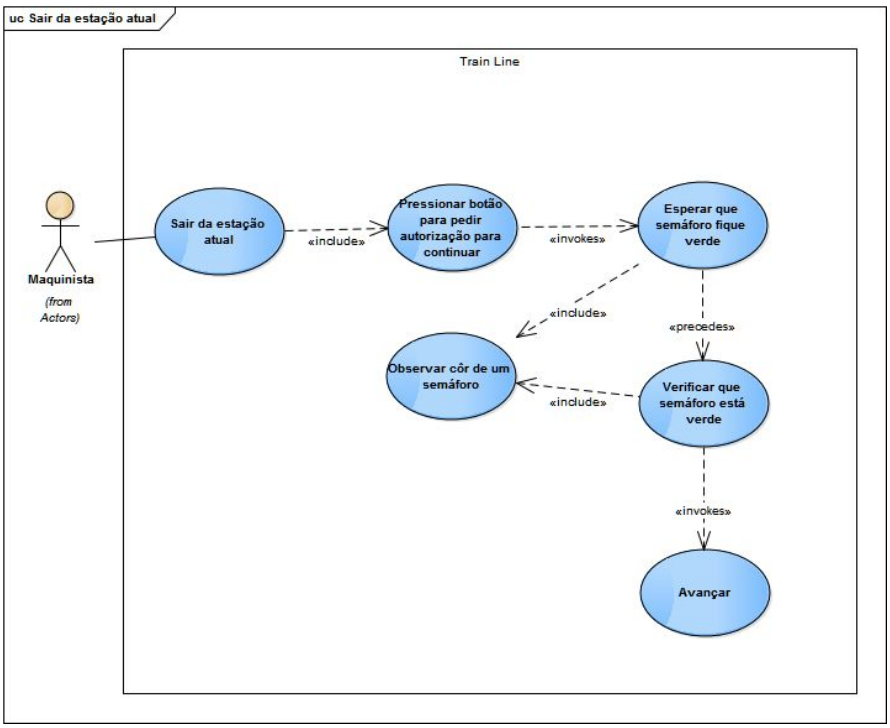

Fig. 7 - Caso de utilização correspondente ao sair de um bloco de uma estação para um bloco simples.

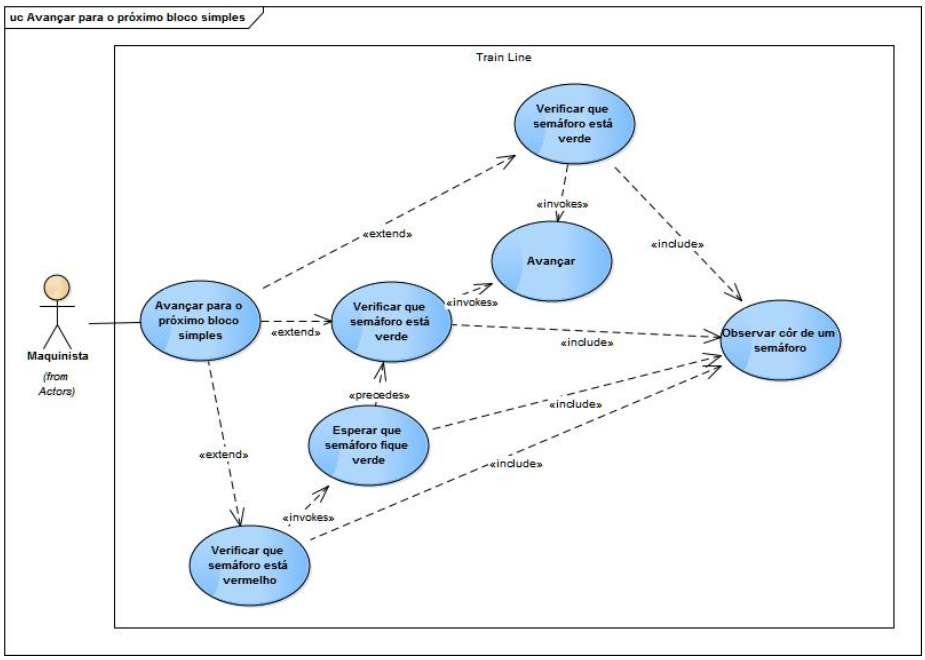

Fig. 8 - Caso de utilização correspondente ao sair de um bloco simples e avançar para outro bloco simples.

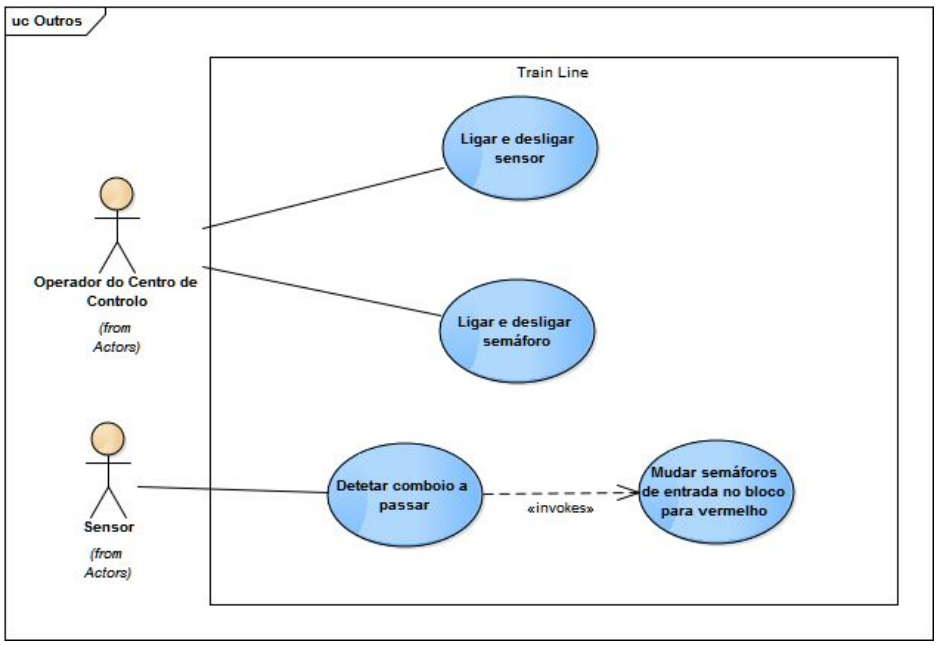

Fig. 9 - Casos de utilização para os atores “Operador de Centro de Controlo” e “Sensor”.

Os casos de utilização encontram-se descritos textualmente nas seguintes tabelas:

2.1.1 - Casos de utilização do Maquinista

Caso	**Avançar para o bloco da estação**
Descrição	Cenário normal de entrada numa estação e

	paragem do veículo nesta.
Pré-condições	1 - O comboio tem de estar numa posição anterior à de uma estação; 2 - O semáforo de entrada no bloco da esquerda da estação não pode estar vermelho nem indisponível. 3 - O sensor do semáforo de entrada no bloco não pode estar indisponível.
Pós-condições	1 - O comboio passa a estar no bloco da esquerda da estação; 2 - O semáforo de entrada no bloco da esquerda da estação está vermelho. 3 - Os semáforos do último e penúltimo blocos obedecem aos requisitos R12 e R13. 4 - O comboio passa a estar no estado 'parado'.
Passos	1 - Avançar para o bloco da estação. 2 - Mudar o estado do comboio para 'parado'.
Excepções	(não especificado)

Caso	**Sair de um bloco de estação**
Descrição	Cenário normal de saída de um comboio de uma estação.
Pré-condições	1 - O comboio tem de estar no estado 'em movimento'. 2 - O comboio tem de estar num bloco da estação. 3 - Não pode existir nenhum sensor ocupado ou indisponível em nenhum bloco até à próxima estação (inclusive).
Pós-condições	1 - O comboio passa a estar no bloco seguinte ao da estação. 2 - O semáforo de entrada no bloco a seguir à estação está vermelho. 3 - Os semáforos do último e penúltimo blocos obedecem aos requisitos R12 e R13. 4 - O comboio passa a estar no estado 'em movimento'.
Passos	1 - Pedir permissão ao centro de controlo para abandonar a estação. 2 - Avançar para o próximo bloco.
Excepções	1 - A permissão para abandonar a estação é negada pelo centro de controlo (passo 1).

Caso	Passagem para de um bloco simples para outro
Descrição	Cenário normal de passagem de um comboio de um bloco simples para outro bloco simples.
Pré-condições	1 - O semáforo de entrada no bloco da esquerda da estação não pode estar vermelho nem indisponível. 2 - O sensor do semáforo de entrada no bloco não pode estar indisponível.
Pós-condições	1 - O veículo passa a estar no novo bloco. 2 - O semáforo do bloco atual está vermelho. 3 - Os semáforos do último e penúltimo blocos obedecem aos requisitos R12 e R13.
Passos	(não especificado)
Excepções	(não especificado)

2.1.2 - Casos de utilização do Operador do Centro de Controlo

Caso	Ligar/Desligar semáforo
Descrição	Cenário normal relativo ao ato de ligar ou desligar um semáforo.
Pré-condições	(não especificado)
Pós-condições	1 - O semáforo fica com o estado desejado no *input*.
Passos	(não especificado)
Excepções	(não especificado)

Caso	Ligar/Desligar sensor do semáforo
Descrição	Cenário normal relativo ao ato de ligar ou desligar um sensor associado a um semáforo.
Pré-condições	(não especificado)
Pós-condições	1 - O sensor fica com o estado desejado no *input*.
Passos	(não especificado)
Excepções	(não especificado)

2.1.3 - Casos de utilização do Sensor

Caso	Alterar semáforo
Descrição	Cenário normal relativo ao facto de ligar ou desligar um sensor associado a um semáforo.
Pré-condições	1 - O sensor está disponível. 2 - O semáforo está disponível.
Pós-condições	1 - O semáforo fica com um estado congruente com os requisitos R12 e R13.
Passos	(não especificado)
Excepções	(não especificado)

2.2 - Classes

2.2.1 - *TrainLine*

A classe *TrainLine* corresponde a uma linha ferroviária circular. Esta apresenta um conjunto de *Modules* e um conjunto de *Trains*.

2.2.2 - *Module*

A classe *Module* corresponde a uma zona de uma linha ferroviária, podendo ser um *SimpleModule* (caso não possua nenhuma estação) ou um *StationModule* (caso possua uma estação). Esta classe tem como variáveis de instância referências para os *modules* imediatamente antes e depois, assim como um identificador.

2.2.3 - *SimpleModule*

Esta classe herda de *Module* e contém como variável de instância um só *Block*.

2.2.4 - *StationModule*

Esta classe herda de *Module* e contém como variáveis de instância dois *Blocks* - dado que uma estação é constituída por dois blocos paralelos.

2.2.5 - *Block*

Esta classe existe no contexto de um *Module*, e corresponde a uma zona delimitada da linha onde apenas pode estar um comboio de cada vez. Este possui também dois semáforos que controlam a entrada de comboios no bloco em cada um dos sentidos possíveis. É esta classe que contém a lógica de alteração dos semáforos quando um comboio sai de um bloco e entra noutro, assim como a lógica de permissão de entrada de um comboio num bloco.

2.2.6 - *Semaphore*

Esta classe existe no contexto de um *Block* e corresponde a um semáforo que se encontra pouco antes deste *Block* e que controla a entrada de comboios neste. Este pode ter uma *Color* (*Green, Yellow* e *Red*) como estado. Possui também um *Sensor* que muda o semáforo quando um comboio passa por este. Quer o semáforo, quer o sensor podem estar indisponíveis.

2.2.7 - *Train*

Esta classe corresponde a um comboio. Esta classe contém a lógica de movimentação dos comboios (que chama a lógica da *Block*) e dos pedidos para sair de uma estação.

2.2.8 - *TrainLineBuilder*

Esta classe é responsável por gerar *TrainLines* predefinidas para efeitos de teste.

2.2.9 - *TestTrainLine*

Esta classe é responsável por albergar os testes da aplicação, descendendo da classe de testes fornecida na disciplina (“TestCase”).

Apresenta-se na *Figura 10* o diagrama de classes da implementação do modelo.

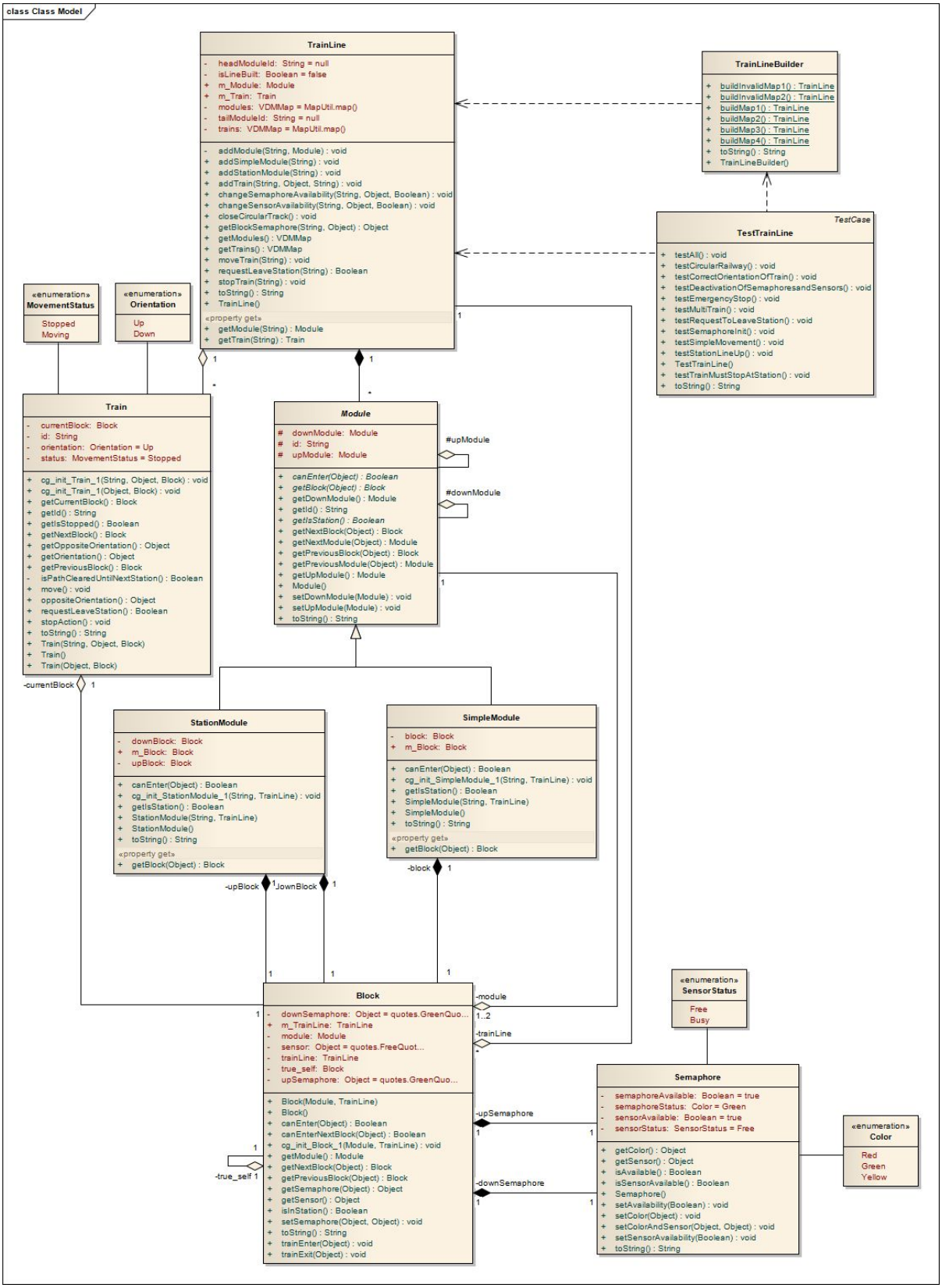

Fig. 10 - Diagrama UML de classes da implementação do sistema.

3 - Modelo Formal em VDM++

3.1 - *TrainLine*

```
class TrainLine
types
        public String = seq of char;
values

instance variables
        private modules: map String to Module := {|->};
        private trains: map String to Train := {|->};

        private headModuleId: [String] := nil;
        private tailModuleId: [String] := nil;
        private isLineBuilt: bool := false;

        -- headModuleId and tailModuleId must be both null before any modules being added
        inv (headModuleId = nil and tailModuleId = nil) => card dom modules = 0;
        inv isLineBuilt = true => (headModuleId <> nil and tailModuleId <> nil);

        -- There arent any trains in the same block
        inv isLineBuilt => not exists t1, t2 in set dom trains & (t1 <> t2 and
trains(t1).getCurrentBlock() = trains(t2).getCurrentBlock());

        -- There must be a minimum of 2 stations
        inv isLineBuilt => exists s1, s2 in set dom modules & (s1 <> s2 and
modules(s1).getIsStation() and modules(s2).getIsStation());

        -- There must be at least 1 module between stations and at least 2 stations
        inv isLineBuilt => forall m in set dom modules &
                (modules(m).getIsStation() =>
                        modules(m).getNextBlock(<Up>).isInStation() = false
                and modules(m).getNextBlock(<Down>).isInStation() = false)

operations

        -- Adds a simple module (between 2 stations) to the line:
        public addSimpleModule: (String) ==> ()
        addSimpleModule(id) ==
        (
                dcl simpleModule: Module := new SimpleModule(id, self);
                addModule(id, simpleModule);
        )
        pre isLineBuilt = false;

        -- Adds a station module to the line:
        public addStationModule: (String) ==> ()
        addStationModule(id) ==
        (
                dcl stationModule: Module := new StationModule(id, self);
                addModule(id, stationModule);
        )
        pre isLineBuilt = false;

        -- Closes the circular track:
  public closeCircularTrack: () ==> ()
        closeCircularTrack() ==
        (
                -- Close the circular track:
         modules(headModuleId).setDownModule(modules(tailModuleId));
         modules(tailModuleId).setUpModule(modules(headModuleId));

                -- The semaphores of simple modules in the station must be red or yellow
                for all id in set dom modules do
```

```
                (
                dcl module: Module := modules(id);

                if module.getIsStation() then
                (
                        -- In the station all sempahores are yellow:
                        module.getBlock(<Down>).setSemaphore(<Yellow>, <Down>);
                        module.getBlock(<Up>).setSemaphore(<Yellow>, <Up>);

                        -- If we leave the station, we have to face a red semaphore:
                 module.getDownModule().getBlock(<Down>).setSemaphore(<Red>, <Down>);
                 module.getUpModule().getBlock(<Up>).setSemaphore(<Red>, <Up>);
                );

                );

                -- The line is now built:
                isLineBuilt := true;
        )
        pre isLineBuilt = false and card dom modules >= 2 -- To close a circular track we
have to have at least two modules in the track
        post modules(headModuleId).getDownModule() = modules(tailModuleId) and -- the
first module must be linked to the last model
                modules(tailModuleId).getUpModule() = modules(headModuleId); -- The last
module must be linked to the first model

        -- Adds a train to the train line:
        public addTrain: (String) * (Train`Orientation) * (String) ==> ()
        addTrain(id, orientation, moduleId) ==
        (
                trains := trains ++ {id |-> new Train(id, orientation,
modules(moduleId).getBlock(orientation))};

modules(moduleId).getBlock(orientation).getPreviousBlock(orientation).trainExit(orientat
ion);
         modules(moduleId).getBlock(orientation).trainEnter(orientation);
        )
        pre isLineBuilt = true and
        id not in set dom trains and
        moduleId in set dom modules and
        modules(moduleId).getIsStation() and -- We can only put new trains in stations
        not exists t1 in set dom trains & trains(t1).getCurrentBlock().getModule() =
modules(moduleId) -- The station must be empty
        post id in set dom trains and
         trains(id).getCurrentBlock().getSemaphore(orientation).getColor() = <Red> and
trains(id).getCurrentBlock().getSemaphore(orientation).getSensor() = <Busy> and -- The
block is occupied and busy
                trains(id).getCurrentBlock().getModule() = modules(moduleId);

        -- Moves a train with a specific id
        public moveTrain: (String) ==> ()
        moveTrain(id) == trains(id).move()
        pre isLineBuilt = true and id in set dom trains;

        -- Stop train in a simple block
        public stopTrain: (String) ==> ()
        stopTrain(id) == trains(id).stopAction()
        pre isLineBuilt = true and id in set dom trains;

        -- Requests a specific train to leave a station
        public requestLeaveStation: (String) ==> bool
        requestLeaveStation(id) == trains(id).requestLeaveStation()
        pre isLineBuilt = true and id in set dom trains;

        -- Changes the availability of a semaphore in a specific module and with a
specific orientation
        public changeSemaphoreAvailability: String * Train`Orientation * bool ==> ()
        changeSemaphoreAvailability(moduleId, orientation, isAvailable) ==
modules(moduleId).getBlock(orientation).getSemaphore(orientation).setAvailability(isAvai
lable)
```

```
        pre moduleId in set dom modules;

        -- Changes the availability of a sensor in a specific module and with a specific
orientation
        public changeSensorAvailability: String * Train`Orientation * bool ==> ()
        changeSensorAvailability(moduleId, orientation, isAvailable) ==
modules(moduleId).getBlock(orientation).getSemaphore(orientation).setSensorAvailability(
isAvailable)
        pre moduleId in set dom modules;

        -- Adds a module to the station
        private addModule: (String) * (Module) ==> ()
        addModule(newModuleId, newModule) ==
        (
                if card dom modules > 0 -- if we already have a module added
                then
                (
                dcl tailModule: Module := modules(tailModuleId);

                -- Connect the new model to the last model insered:
                        tailModule.setUpModule(newModule);
                        newModule.setDownModule(tailModule);

                -- Update the last model inserted:
                tailModuleId := newModuleId;
                )
                else
                (
                atomic
                (
                        headModuleId := newModuleId;
                        tailModuleId := newModuleId;
                );
                );

                modules := modules munion { newModuleId |-> newModule };
        )
        pre isLineBuilt = false and newModuleId not in set dom modules
        post newModuleId in set dom modules and -- The id was added to the modules
          if card dom modules~ = 0
                then (headModuleId = newModuleId and tailModuleId = newModuleId)
                else (tailModuleId = newModuleId and modules(tailModuleId~).getUpModule()
= newModule and newModule.getDownModule() = modules(tailModuleId~));

        -- Gets a train by its identifier
        public pure getTrain: String ==> Train
        getTrain(id) == return trains(id);

        -- Gets a module by its identifier
        public pure getModule: String ==> Module
        getModule(id) == return modules(id);

        -- Gets the semaphore in a block insired in a module with a specific orientation
        public pure getBlockSemaphore: String * Train`Orientation ==> Semaphore`Color
        getBlockSemaphore(idArg, orientationArg) ==
        (

getModule(idArg).getBlock(orientationArg).getSemaphore(orientationArg).getColor();
        );

        -- Gets the map of trains
        pure public getTrains: () ==> (map String to Train)
        getTrains() ==
        (
                return trains;
        );

        -- Gets the map of modules
        pure public getModules: () ==> (map String to Module)
        getModules() ==
```

```
        (
                return modules;
        );

end TrainLine
```

3.2 - *Module*

```
class Module

instance variables
        protected id: TrainLine`String; -- Identifier of this Module
        protected upModule: Module; -- module this module will reach if going in the 'Up'
direction
        protected downModule: Module; -- module this module will reach if going in the
'Down' direction

operations

  -- Gets the block in a specific orientation
  -- => In a SimpleModule, there is only one block to be shared by both Train
Orientations
  -- => In a StationModuel, there are two blocks to be used from trains in different
Train Orientations
        public pure getBlock: (Train`Orientation) ==> (Block)
        getBlock(orientation) ==
        is subclass responsibility;

        -- Gets the next block in a specific orientation
        public pure getNextBlock: (Train`Orientation) ==> (Block)
        getNextBlock(orientation) ==
        (
                if orientation = <Up>
                then
                return upModule.getBlock(orientation)
                else
                return downModule.getBlock(orientation);
        );

         -- Gets the previous block in a specific orientation
        public pure getPreviousBlock: (Train`Orientation) ==> (Block)
        getPreviousBlock(orientation) ==
        (
                if orientation = <Down>
                then
                return upModule.getBlock(orientation)
                else
                return downModule.getBlock(orientation);
        );

        -- Gets the identifier of this module
        public pure getId: () ==> TrainLine`String
        getId() == return id;

        -- Checks if this module is a station or not
        public pure getIsStation: () ==> bool
        getIsStation() ==
        is subclass responsibility;

        -- Gets the next module in the "Up" train orientation
        public pure getUpModule: () ==> Module
        getUpModule() == return upModule;

        -- Gets the previous module in the <Down> train orientation
        public pure getDownModule: () ==> Module
        getDownModule() == return downModule;
```

```
        -- Gets the next module, according to a specific orientation
        pure public getNextModule: (Train`Orientation) ==> (Module)
        getNextModule(orientation) ==
        (
                if orientation = <Up>
                then return upModule else return downModule;
        );

        -- Gets the previous module, according to a specific orientation
        pure public getPreviousModule: (Train`Orientation) ==> (Module)
        getPreviousModule(orientation) ==
        (
                if orientation = <Down>
                then return upModule else return downModule;
        );

        -- Sets the module in the <Up> train orientation
        public setUpModule: (Module) ==> ()
        setUpModule(upModuleArg) == upModule := upModuleArg;

        -- Sets the module in the <Down> train orientation
        public setDownModule: (Module) ==> ()
        setDownModule(downModuleArg) == downModule := downModuleArg;

end Module
```

3.3 - *SimpleModule*

```
class SimpleModule is subclass of Module

instance variables
        private block: Block; -- A simple module has one single block

operations
  -- Constructor
        public SimpleModule: (TrainLine`String) * (TrainLine) ==> SimpleModule
        SimpleModule(idArg, trainLineArg) ==
        (
                id := idArg;
                block := new Block(self, trainLineArg);
                return self;
        );

        -- A simple module has one single block to be shared in both train orientations,
so it ignores the train orientation
        public pure getBlock: (Train`Orientation) ==> (Block)
        getBlock(orientation) ==
        (
                return block;
        );

        public pure getIsStation: () ==> bool
        getIsStation() == return false;

end SimpleModule
```

3.4 - *StationModule*

```
class StationModule is subclass of Module

instance variables
        private upBlock: Block; -- A station module has 2 possible blocks for a train to
```

```
be one, depending on its direction
        private downBlock: Block;

operations
  -- Constructor
        public StationModule: (TrainLine`String) * (TrainLine) ==> StationModule
        StationModule(idArg, trainLineArg) ==
        (
                id := idArg;
                upBlock := new Block(self, trainLineArg);
                downBlock := new Block(self, trainLineArg);
                return self;
        );

        public pure getBlock: (Train`Orientation) ==> (Block)
        getBlock(orientation) ==
        (
                if orientation = <Up>
                then
                return upBlock else return downBlock;
        );

        public pure getIsStation: () ==> bool
        getIsStation() == return true;

end StationModule
```

3.5 - *Block*

```
class Block

instance variables
        private module : Module; -- Module in which this Block is insired
        private trainLine: TrainLine; -- Trainline in which this block is insired
        private upSemaphore: Semaphore := new Semaphore(); -- Semaphore that controls the
entries in the Up direction
        private downSemaphore: Semaphore := new Semaphore(); -- Semaphore that controls
the entries in the Down direction

operations

        -- Constructor
        public Block: (Module) * (TrainLine) ==> Block
        Block(moduleArg, trainLineArg) ==
        (
                module := moduleArg;
                trainLine := trainLineArg;
                return self;
        );

        -- Makes the modifications in this block's semaphores as if a train is entering
this block
        public trainEnter: (Train`Orientation) ==> ()
        trainEnter(trainOrientation) ==
        (
                if trainOrientation = <Up>
                then
                        upSemaphore.setColorAndSensor(<Red>, <Busy>)
                else
                        downSemaphore.setColorAndSensor(<Red>, <Busy>);
        );

        -- Makes the modifications in this block's semaphores as if a train is exiting
this block
        public trainExit: (Train`Orientation) ==> ()
        trainExit(trainOrientation) ==
        (
                -- Update the block we're leaving
```

```
                dcl semaphoreNextColor: Semaphore`Color;
                dcl previousSemaphoreNextColor: Semaphore`Color;

                -- If this is a module after a station, it must be red at all time,
otherwise yellow
                if getModule().getPreviousModule(trainOrientation).getIsStation()
                then
                semaphoreNextColor := <Red>
                else
                semaphoreNextColor := <Yellow>;

                if trainOrientation = <Up>
                then
                 upSemaphore.setColorAndSensor(semaphoreNextColor, <Free>)
                else
                 downSemaphore.setColorAndSensor(semaphoreNextColor, <Free>);

                -- Check the previous block to the block we're leaving:
                -- If the previous-previous block is a station or if a train is there,
then it must be red
                if
getPreviousBlock(trainOrientation).getPreviousBlock(trainOrientation).isInStation()
                        or
getPreviousBlock(trainOrientation).getSemaphore(trainOrientation).getSensor() = <Busy>
then
                previousSemaphoreNextColor := <Red>
                elseif getPreviousBlock(trainOrientation).isInStation() then -- If the
previous block from the block we're leaving is a station, leave it yellow
                previousSemaphoreNextColor := <Yellow>
                else
                previousSemaphoreNextColor := <Green>;

                -- Update the semalhore:

getModule().getPreviousModule(trainOrientation).getBlock(trainOrientation).setSemaphore(
previousSemaphoreNextColor, trainOrientation);
        );

        -- Checks if it's safe for a train to enter this block in this direction
(orientation)
        pure public canEnter: (Train`Orientation) ==> (bool)
        canEnter(orientationArg) ==
                if orientationArg = <Up>
                then
                return upSemaphore.getColor() <> <Red> and upSemaphore.isAvailable()
                else
                return downSemaphore.getColor() <> <Red> and downSemaphore.isAvailable();

        -- Checks if this block is insired in a station module
        pure public isInStation: () ==> (bool)
        isInStation() == return module.getIsStation();

        -- Gets the semaphore from this block, according to a certain orientation
        pure public getSemaphore: (Train`Orientation) ==> (Semaphore)
        getSemaphore(orientationArg) ==
                if orientationArg = <Up>
                then
                return upSemaphore else
                return downSemaphore;

        -- Gets the module in which this block is insired
        pure public getModule: () ==> (Module)
        getModule() == return module;

        -- Gets the next block to this block, in a specified orientation
        pure public getNextBlock: (Train`Orientation) ==> (Block)
        getNextBlock(orientation) == return module.getNextBlock(orientation);

        -- Gets the previous block to this block, in a specified orientation
        pure public getPreviousBlock: (Train`Orientation) ==> (Block)
```

```
        getPreviousBlock(orientation) == return module.getPreviousBlock(orientation);

        -- Sets the value of the semaphore that controls the traffic coming from a
specific orientation
        public setSemaphore: (Semaphore`Color) * (Train`Orientation) ==> ()
        setSemaphore(semaphoreArg, orientationArg) ==
                if orientationArg = <Up>
                then
                        upSemaphore.setColor(semaphoreArg)
                else
                        downSemaphore.setColor(semaphoreArg);

end Block
```

3.6 - *Semaphore*

```
class Semaphore
types
        public Color = <Red> | <Yellow> | <Green>; -- Possible state colors for the
Semaphore
        public SensorStatus = <Free> | <Busy>; -- Possible status for the Sensor

instance variables
        private semaphoreStatus: Color := <Green>; -- The current color of the semaphore
        private sensorStatus: SensorStatus := <Free>; -- The current status for the
sensor
        private semaphoreAvailable: bool := true; -- Flag that indicates if a semaphore
is available
        private sensorAvailable: bool := true; -- Flag that indicates if a sensor is
available

        -- If the current sensor is busy, then the semaphore should always be red
        inv sensorStatus = <Busy> => semaphoreStatus = <Red>;

operations

        public pure getColor: () ==> Color
        getColor() == return semaphoreStatus;

        public pure isAvailable: () ==> bool
        isAvailable() == return semaphoreAvailable and sensorAvailable;

        public pure getSensor: () ==> SensorStatus
        getSensor() == return sensorStatus;

        public pure isSensorAvailable: () ==> bool
        isSensorAvailable() == return sensorAvailable;

        public setAvailability: bool ==> ()
        setAvailability(availability) == semaphoreAvailable := availability;

        public setSensorAvailability: bool ==> ()
        setSensorAvailability(availability) == sensorAvailable := availability;

        public setColor: (Color) ==> ()
        setColor(color) == semaphoreStatus := color;

        public setColorAndSensor: Color * SensorStatus ==> ()
        setColorAndSensor(color, sensor) ==
        atomic
        (
                semaphoreStatus := color;
                sensorStatus := sensor;
        );

end Semaphore
```

3.7 - *Train*

```
class Train
types
  -- Possible orientations for a train:
        public Orientation = <Up> | <Down>;
        -- Possible status for a train:
        public MovementStatus = <Stopped> | <Moving>;
values

instance variables
        private id: TrainLine`String; -- Train identifier

        private orientation: Orientation := <Up>; -- Orientation of this train
        private status: MovementStatus := <Stopped>; -- Current status of this train
        private currentBlock: Block; -- Current block of the train

operations
        -- Constructor
        public Train: (TrainLine`String) * (Orientation) * (Block) ==> Train
        Train(idArg, initialOrientation, initialBlock) ==
        (
                id := idArg;
                orientation := initialOrientation;
                currentBlock := initialBlock;
                return self;
        );

        -- Moves the train to the next block
        public move: () ==> ()
        move() ==
        (
                -- Update status:
                status := <Moving>;

                -- Make exit operations on the block we were on:
                currentBlock.trainExit(orientation);

                -- Make enter operations on the block we're going on:
                getNextBlock().trainEnter(orientation);

                -- Move to the next block:
                currentBlock := getNextBlock();
        )
        pre getNextBlock().canEnter(orientation)
        post currentBlock = currentBlock~.getNextBlock(orientation) and -- New block is
equal to old block's next block
                not currentBlock.canEnter(orientation) and -- We cannot enter in the new
block (New block is red)
           (if getPreviousBlock().getPreviousBlock(orientation).isInStation() then --
Check if sempahore from the block we're exiting is ok
                 getPreviousBlock().getSemaphore(orientation).getColor() = <Red>
                else
                 getPreviousBlock().getSemaphore(orientation).getColor() = <Yellow>
           ) and
           (if
getPreviousBlock().getPreviousBlock(orientation).getPreviousBlock(orientation).isInStati
on()
                        or
getPreviousBlock().getPreviousBlock(orientation).getSemaphore(orientation).getSensor() =
<Busy> then -- We should put red if its a block next to a station, or a train is there

getPreviousBlock().getPreviousBlock(orientation).getSemaphore(orientation).getColor() =
<Red>
                elseif getPreviousBlock().getPreviousBlock(orientation).isInStation() then

getPreviousBlock().getPreviousBlock(orientation).getSemaphore(orientation).getColor() =
<Yellow>
```

```
                else

getPreviousBlock().getPreviousBlock(orientation).getSemaphore(orientation).getColor() =
<Green>
          );

        -- Stops the train
        public stopAction: () ==> ()
        stopAction() == status := <Stopped>
        pre status = <Moving>;

        -- Requests the train to leave the current station
        public requestLeaveStation: () ==> bool
        requestLeaveStation() ==
        (
                -- See if we can exit ==> the set of blocks until the next station must be
cleared and the next semaphore must work
                if isPathClearedUntilNextStation() and
getNextBlock().getSemaphore(orientation).isAvailable()
                then
                (
                        getNextBlock().setSemaphore(<Green>, orientation);
                return true;
                )
                else
                return false;
        )
        pre status = <Stopped> and currentBlock.isInStation() and
getNextBlock().getSemaphore(orientation).getColor() = <Red>
        post if isPathClearedUntilNextStation() and
getNextBlock().getSemaphore(orientation).isAvailable() then
                        getNextBlock().getSemaphore(orientation).getColor() = <Green>
                else getNextBlock().getSemaphore(orientation).getColor() = <Red>;

        -- Gets the current block in which the train is
        pure public getCurrentBlock: () ==> Block
        getCurrentBlock() == return currentBlock;

        -- Gets the next block the train is going to occupy according to his orientation
        pure public getNextBlock: () ==> Block
        getNextBlock() == return currentBlock.getNextBlock(orientation);

        -- Gets the previous block the train is going to occupy according to his
orientation
        pure public getPreviousBlock: () ==> Block
        getPreviousBlock() == return currentBlock.getPreviousBlock(orientation);

        -- Gets the train's orientation
        pure public getOrientation: () ==> Orientation
        getOrientation() == return orientation;

        -- Checks if a train is stopped at the moment
        pure public getIsStopped: () ==> bool
        getIsStopped() == return status = <Stopped>;

        -- Gets the train's identifier
        pure public getId: () ==> TrainLine`String
        getId() == return id;

        -- Checks if the path from a specific station to the next one is clear, as in, no
sensors is occupied during the way
        private pure isPathClearedUntilNextStation: () ==> bool
        isPathClearedUntilNextStation() ==
        (
                dcl nextBlock: Block := getNextBlock();

                -- Iterate the next blocks and check if we can make it to the next station
without being stuck somewhere in between:
                while (not nextBlock.getModule().getIsStation())
                and nextBlock.getSemaphore(orientation).getSensor() = <Free> and
```

```
nextBlock.getSemaphore(orientation).isSensorAvailable()
                and nextBlock.getSemaphore(oppositeOrientation()).getSensor() = <Free> and
nextBlock.getSemaphore(oppositeOrientation()).isSensorAvailable() do
                        nextBlock := nextBlock.getNextBlock(orientation);

                -- If we got to the next station in the previous cycle, then the path is
clear, but we have to check if it's busy
                -- But in the station, we only want to check if the semaphore for our
orientaton is on
                return (nextBlock.getModule().getIsStation() and
nextBlock.getSemaphore(orientation).getSensor() = <Free> and
nextBlock.getSemaphore(orientation).isSensorAvailable());
        )
        pre status = <Stopped> and currentBlock.isInStation(); -- We can only execute
this function when we're stopped in a station

        -- Returns the opposite orientation to the train's orientation
        public pure oppositeOrientation: () ==> (Orientation)
        oppositeOrientation() ==
        (
                if orientation = <Up>
                then return <Down>
                else return <Up>;
        );

end Train
```

4 - Validação do Modelo

Apresenta-se de seguida o código de teste do modelo. Em comentário em cima das operações estão indicados os requisitos que estão a ser testados. Considera-se que, por serem mais de nível estrutural das classes do modelo, os requisitos R2, R5, R6, R7 e R19 não têm associados diretamente um teste unitário.

4.1 - *TestTrainLine*

```
class TestTrainLine is subclass of TestCase

operations

        -- Tests requirement R1
        public testCircularRailway: () ==> ()
        testCircularRailway() ==
        (
                dcl line: TrainLine := TrainLineBuilder`buildMap3();

                assertEqual(line.getModule("station1"),
line.getModule("module1").getUpModule());
                assertEqual(line.getModule("module1"),
line.getModule("station1").getDownModule());
                assertEqual(line.getModule("station1"),
line.getModule("module1").getNextModule(<Up>));
                assertEqual(line.getModule("module1"),
line.getModule("station1").getNextModule(<Down>));
                assertEqual("module1", line.getModule("module1").getId());

                assertEqual(line.getModule("module2"),
line.getModule("station1").getUpModule());
                assertEqual(line.getModule("station1"),
line.getModule("module2").getDownModule());
```

```
                assertEqual(line.getModule("station2"),
line.getModule("module2").getUpModule());
                assertEqual(line.getModule("module2"),
line.getModule("station2").getDownModule());

                assertEqual(line.getModule("module1"),
line.getModule("station2").getUpModule());
                assertEqual(line.getModule("station2"),
line.getModule("module1").getDownModule());

                assertEqual(line.getTrains(), line.getTrains());
                assertEqual(line.getModules(), line.getModules());

                IO`println("Passed test circular railway test");
        );

        -- Tests requirements R3 and R4
        public testStationLineUp: () ==> ()
        testStationLineUp() ==
        (
                -- **** Comment in order to pass ****
                --dcl map1: TrainLine := TrainLineBuilder`buildInvalidMap1();
                --dcl map2: TrainLine := TrainLineBuilder`buildInvalidMap2();

                IO`println("Passed test station line up test");
        );

        -- Tests requirements R8
        public testTrainMustStopAtStation: () ==> ()
        testTrainMustStopAtStation() ==
        (
                dcl line: TrainLine := TrainLineBuilder`buildMap1();
                assertEqual("train1", line.getTrain("train1").getId());

                assertEqual(true, line.requestLeaveStation("train1"));
                line.moveTrain("train1");
                line.moveTrain("train1");
                line.moveTrain("train1");
                line.moveTrain("train1");

                line.stopTrain("train1"); -- Uncomment in order to pass ==> we must call
stopTrain before we ask for permition to leave

                assertEqual(true, line.requestLeaveStation("train1"));

                IO`println("Passed test must stop at station");
        );

        -- Tests requirement R9
        public testCorrectOrientationOfTrain: () ==> ()
        testCorrectOrientationOfTrain() ==
        (
                dcl line: TrainLine := TrainLineBuilder`buildMap1();
                dcl train1: Train := line.getTrain("train1");

                assertEqual(true, line.requestLeaveStation("train1"));
                line.moveTrain("train1");
                line.moveTrain("train1");
                line.moveTrain("train1");
                line.moveTrain("train1");

                line.stopTrain("train1");

                assertEqual(train1.getCurrentBlock(),
line.getModule("station2").getBlock(train1.getOrientation()));
                IO`println("Passed correct of orientation of a train in a station");
        );

        -- Tests requirements R10 and R11 (only tests the sensor busy situation)
```

```
        public testRequestToLeaveStation: () ==> ()
        testRequestToLeaveStation() ==
        (
                dcl line: TrainLine := TrainLineBuilder`buildMap2();

                assertEqual(true, line.requestLeaveStation("train1"));
                line.moveTrain("train1");

                assertEqual(false, line.requestLeaveStation("train2"));
                assertEqual(false, line.requestLeaveStation("train3"));

                IO`println("Passed request to leave station test");
        );

        -- Test emergency stops (R11 - only busy situation - and R15) and movement (R14)
        public testEmergencyStop: () ==> ()
        testEmergencyStop() ==
        (
                dcl line: TrainLine := TrainLineBuilder`buildMap2();

                assertEqual(true, line.requestLeaveStation("train2"));
                line.moveTrain("train2");
                line.moveTrain("train2");
                line.moveTrain("train2");
                line.stopTrain("train2");
                assertEqual(line.getModule("module2"),
line.getTrain("train2").getCurrentBlock().getModule());

                assertEqual(true, line.requestLeaveStation("train3"));
                line.moveTrain("train3");
                line.moveTrain("train3");
                line.moveTrain("train3");
                line.stopTrain("train3");
                assertEqual(false, line.requestLeaveStation("train3"));

                IO`println("Passed emergency stop test");
        );

        -- Tests requirements R11 (situation where sensors are not available), R17, R18
        public testDeactivationOfSemaphoresandSensors: () ==> ()
        testDeactivationOfSemaphoresandSensors() ==
        (
                dcl line: TrainLine := TrainLineBuilder`buildMap4();

                line.changeSensorAvailability("module2",
line.getTrain("train2").getOrientation(), false);
                assertEqual(false, line.requestLeaveStation("train2"));

                line.changeSensorAvailability("module2",
line.getTrain("train2").getOrientation(), true);
                assertEqual(true, line.requestLeaveStation("train2"));

                line.changeSemaphoreAvailability("module2",
line.getTrain("train2").getOrientation(), false);
                line.moveTrain("train2");
                line.moveTrain("train2");
                --line.moveTrain("train2"); -- *** Comment to pass => we cannot move ***

                line.changeSemaphoreAvailability("module2",
line.getTrain("train2").getOrientation(), true);
                line.moveTrain("train2");

                line.changeSemaphoreAvailability("station1",
line.getTrain("train2").getOrientation(), false);
                --line.moveTrain("train2"); -- *** Comment to pass => we cannot move ***
                line.changeSemaphoreAvailability("station1",
line.getTrain("train2").getOrientation(), true);
                line.moveTrain("train2");
                line.stopTrain("train2");
```

```
                -- In the station, if the next semaphore is not working, we cant advance
                line.changeSemaphoreAvailability("module1",
line.getTrain("train2").getOrientation(), false);
                assertEqual(false, line.requestLeaveStation("train2"));
                line.changeSemaphoreAvailability("module1",
line.getTrain("train2").getOrientation(), true);
                assertEqual(true, line.requestLeaveStation("train2"));

                -- If a sensor is not working, a semaphore is also not working
                line.changeSensorAvailability("module7",
line.getTrain("train2").getOrientation(), false);
                line.moveTrain("train2");
                line.moveTrain("train2");
                line.moveTrain("train2");
                line.stopTrain("train2");

                assertEqual(false, line.requestLeaveStation("train2"));
                line.changeSensorAvailability("module7",
line.getTrain("train2").getOrientation(), true);
                assertEqual(true, line.requestLeaveStation("train2"));

                IO`println("Passed deactivation of semaphores and sensors test");
        );

        -- Tests requirements R12 and R13 (focusing in initializing the semaphores
correctly)
        public testSemaphoreInit: () ==> ()
        testSemaphoreInit() ==
        (
                dcl line: TrainLine := TrainLineBuilder`buildMap1();
                dcl orientation: Train`Orientation :=
line.getTrain("train1").getOrientation();
                dcl opOrientation: Train`Orientation :=
line.getTrain("train1").oppositeOrientation();

                -- Station with a train:
                assertEqual(<Red>, line.getBlockSemaphore("station1", orientation));
                assertEqual(<Yellow>, line.getBlockSemaphore("station1", opOrientation));
                assertEqual(<Yellow>, line.getBlockSemaphore("module1", orientation));
                assertEqual(<Red>, line.getBlockSemaphore("module1", opOrientation));
                assertEqual(<Red>, line.getBlockSemaphore("module2", orientation));
                assertEqual(<Green>, line.getBlockSemaphore("module2", opOrientation));

                -- Station without a train:
                assertEqual(<Yellow>, line.getBlockSemaphore("station2", orientation));
                assertEqual(<Yellow>, line.getBlockSemaphore("station2", opOrientation));
                assertEqual(<Green>, line.getBlockSemaphore("module3", orientation));
                assertEqual(<Red>, line.getBlockSemaphore("module3", opOrientation));
                assertEqual(<Red>, line.getBlockSemaphore("module4", orientation));
                assertEqual(<Green>, line.getBlockSemaphore("module4", opOrientation));

                IO`println("Passed semaphore initialization");
        );

        -- Tests requirements R12 and R13 (focusing in simple train movement)
        public testSimpleMovement: () ==> ()
        testSimpleMovement() ==
        (
                dcl line: TrainLine := TrainLineBuilder`buildMap1();
                dcl orientation: Train`Orientation :=
line.getTrain("train1").getOrientation();

                -- Test if the train is in the station, with a red light before the
station and after the station
                assertEqual(line.getTrain("train1").getCurrentBlock(),
line.getModule("station1").getBlock(orientation));
                assertEqual(true, line.getTrain("train1").getIsStopped()); -- Test if we
are stopped
                assertEqual(<Yellow>, line.getBlockSemaphore("module1", orientation));
                assertEqual(<Red>, line.getBlockSemaphore("station1", orientation));
```

```
                assertEqual(<Red>, line.getBlockSemaphore("module2", orientation));

                -- If we ask to leave the station, the semaphore before the station goes
orange and the after goes green
                assertEqual(true, line.requestLeaveStation("train1"));
                assertEqual(<Yellow>, line.getBlockSemaphore("module1", orientation));
                assertEqual(<Red>, line.getBlockSemaphore("station1", orientation));
                assertEqual(<Green>, line.getBlockSemaphore("module2", orientation));
                assertEqual(true, line.getTrain("train1").getIsStopped());

                line.moveTrain("train1");
        assertEqual(line.getTrain("train1").getCurrentBlock(),
line.getModule("module2").getBlock(orientation)); -- Check if we moved
                assertEqual(false, line.getTrain("train1").getIsStopped());
                assertEqual(<Green>, line.getBlockSemaphore("module1", orientation));
                assertEqual(<Yellow>, line.getBlockSemaphore("station1", orientation));
                assertEqual(<Red>, line.getBlockSemaphore("module2", orientation));

                line.moveTrain("train1");
        assertEqual(line.getTrain("train1").getCurrentBlock(),
line.getModule("module2.5").getBlock(orientation)); -- Check if we moved
                assertEqual(false, line.getTrain("train1").getIsStopped());
                assertEqual(<Yellow>, line.getBlockSemaphore("station1", orientation));
                assertEqual(<Red>, line.getBlockSemaphore("module2", orientation));
                assertEqual(<Red>, line.getBlockSemaphore("module2.5", orientation));
                assertEqual(<Green>, line.getBlockSemaphore("module3", orientation));

                line.moveTrain("train1");
        assertEqual(line.getTrain("train1").getCurrentBlock(),
line.getModule("module3").getBlock(orientation));
                assertEqual(false, line.getTrain("train1").getIsStopped());
                assertEqual(<Yellow>, line.getBlockSemaphore("station1", orientation));
                assertEqual(<Red>, line.getBlockSemaphore("module2", orientation));
                assertEqual(<Yellow>, line.getBlockSemaphore("module2.5", orientation));
                assertEqual(<Yellow>, line.getBlockSemaphore("station2", orientation));
                assertEqual(<Red>, line.getBlockSemaphore("module3", orientation));

                line.moveTrain("train1");
                line.stopTrain("train1");
        assertEqual(line.getTrain("train1").getCurrentBlock(),
line.getModule("station2").getBlock(orientation));
                assertEqual(<Yellow>, line.getBlockSemaphore("module3", orientation));
                assertEqual(<Red>, line.getBlockSemaphore("station2", orientation));
                assertEqual(<Red>, line.getBlockSemaphore("module4", orientation));
                assertEqual(true, line.getTrain("train1").getIsStopped());

                assertEqual(true, line.requestLeaveStation("train1"));
                assertEqual(true, line.getTrain("train1").getIsStopped());
                assertEqual(<Green>, line.getBlockSemaphore("module4", orientation));

                line.moveTrain("train1");
                assertEqual(false, line.getTrain("train1").getIsStopped());
                assertEqual(<Yellow>, line.getBlockSemaphore("station2", orientation));
                assertEqual(<Red>, line.getBlockSemaphore("module4", orientation));
                assertEqual(<Green>, line.getBlockSemaphore("module5", orientation));

                IO`println("Passed simple movement test");
        );

        -- Tests requirements R11 (related to permission to leave the station)
        public testMultiTrain: () ==> ()
        testMultiTrain() ==
        (
                dcl line: TrainLine := TrainLineBuilder`buildMap2();

                assertEqual(true, line.requestLeaveStation("train2"));
                line.moveTrain("train2");

                -- Movings for train 3:
                assertEqual(true, line.requestLeaveStation("train3"));
```

```
                line.moveTrain("train3");
                line.moveTrain("train3");
                line.moveTrain("train3");
                line.stopTrain("train3");
                assertEqual(line.getModule("station2"),
line.getTrain("train3").getCurrentBlock().getModule());
                assertEqual(false, line.requestLeaveStation("train3"));
                -- End of movings for train 3, while moving train 2

                line.moveTrain("train2");
                line.moveTrain("train2");
                line.moveTrain("train2");
                line.stopTrain("train2");
                assertEqual(line.getModule("station1"),
line.getTrain("train2").getCurrentBlock().getModule());

                assertEqual(true, line.requestLeaveStation("train1"));
                line.moveTrain("train1");
                line.moveTrain("train1");
                line.moveTrain("train1");
                line.moveTrain("train1");
                line.stopTrain("train1");
                assertEqual(line.getModule("station2"),
line.getTrain("train1").getCurrentBlock().getModule());

                IO`println("Passed multi train test");
        );

  public testAll: () ==> ()
  testAll() ==
  (
        testCircularRailway();
        testStationLineUp();
        testTrainMustStopAtStation();
        testCorrectOrientationOfTrain();
        testRequestToLeaveStation();
        testEmergencyStop();
        testDeactivationOfSemaphoresandSensors();
        testSemaphoreInit();
        testSimpleMovement();
        testMultiTrain();
  );

end TestTrainLine
```

4.2 - TrainLineBuilder

```
class TrainLineBuilder
types

values

instance variables

operations

        public static buildMap1: () ==> TrainLine
        buildMap1() ==
        (
                dcl trainLine: TrainLine := new TrainLine();

                trainLine.addSimpleModule("module1");
                trainLine.addStationModule("station1");
                trainLine.addSimpleModule("module2");
                trainLine.addSimpleModule("module2.5");
                trainLine.addSimpleModule("module3");
                trainLine.addStationModule("station2");
```

```
        trainLine.addSimpleModule("module4");
        trainLine.addSimpleModule("module5");
        trainLine.addStationModule("station3");
        trainLine.addSimpleModule("module6");
        trainLine.addSimpleModule("module7");
        trainLine.addStationModule("station4");
        trainLine.addSimpleModule("module8");
        trainLine.closeCircularTrack();

        trainLine.addTrain("train1", <Up>, "station1");

        return trainLine;
);

public static buildMap2: () ==> TrainLine
buildMap2() ==
(
        dcl trainLine: TrainLine := new TrainLine();

        trainLine.addSimpleModule("module1");
        trainLine.addStationModule("station1");
        trainLine.addSimpleModule("module2");
        trainLine.addSimpleModule("module2.5");
        trainLine.addSimpleModule("module3");
        trainLine.addStationModule("station2");
        trainLine.addSimpleModule("module4");
        trainLine.addSimpleModule("module5");
        trainLine.addStationModule("station3");
        trainLine.addSimpleModule("module6");
        trainLine.addSimpleModule("module7");
        trainLine.addStationModule("station4");
        trainLine.addSimpleModule("module8");
        trainLine.closeCircularTrack();

        trainLine.addTrain("train1", <Up>, "station1");
        trainLine.addTrain("train2", <Down>, "station2");
        trainLine.addTrain("train3", <Down>, "station3");

        return trainLine;
);

public static buildMap3: () ==> TrainLine
buildMap3() ==
(
        dcl trainLine: TrainLine := new TrainLine();

        trainLine.addSimpleModule("module1");
        trainLine.addStationModule("station1");
        trainLine.addSimpleModule("module2");
        trainLine.addStationModule("station2");
        trainLine.closeCircularTrack();

        trainLine.addTrain("train1", <Up>, "station1");

        return trainLine;
);

public static buildMap4: () ==> TrainLine
buildMap4() ==
(
        dcl trainLine: TrainLine := new TrainLine();

        trainLine.addSimpleModule("module1");
        trainLine.addStationModule("station1");
        trainLine.addSimpleModule("module2");
        trainLine.addSimpleModule("module2.5");
        trainLine.addSimpleModule("module3");
        trainLine.addStationModule("station2");
        trainLine.addSimpleModule("module4");
        trainLine.addSimpleModule("module5");
```

```
                trainLine.addStationModule("station3");
                trainLine.addSimpleModule("module6");
                trainLine.addSimpleModule("module7");
                trainLine.addStationModule("station4");
                trainLine.addSimpleModule("module8");
                trainLine.closeCircularTrack();

                trainLine.addTrain("train2", <Down>, "station2");

                return trainLine;
        );

        public static buildInvalidMap1: () ==> TrainLine
        buildInvalidMap1() ==
        (
                dcl trainLine: TrainLine := new TrainLine();

                trainLine.addSimpleModule("module1");
                trainLine.addSimpleModule("module2");
                trainLine.addStationModule("station2");
                trainLine.addStationModule("station3");
                trainLine.closeCircularTrack();

                return trainLine;
        );

        public static buildInvalidMap2: () ==> TrainLine
        buildInvalidMap2() ==
        (
                dcl trainLine: TrainLine := new TrainLine();

                trainLine.addSimpleModule("module1");
                trainLine.addStationModule("station1");
                trainLine.addSimpleModule("module2");
                trainLine.closeCircularTrack();

                return trainLine;
        );

functions

traces


end TrainLineBuilder
```

5 - Verificação do Modelo

5.1 - Exemplo de verificação do domínio

Número	Nome da *Proof Obligation*	Tipo
42	*TrainLine`addTrain(String, Orientation, String)*	*Legal map application*

O código em análise é o seguinte (com o acesso ao *map* a vermelho):

```
        -- Adds a train to the train line:
        public addTrain: (String) * (Train`Orientation) * (String) ==> ()
        addTrain(id, orientation, moduleId) ==
```

```
        (
                trains := trains ++ {id |-> new Train(id, orientation,
modules(moduleId).getBlock(orientation))};

modules(moduleId).getBlock(orientation).getPreviousBlock(orientation).trainExit(orientat
ion);
         modules(moduleId).getBlock(orientation).trainEnter(orientation);
        )
        pre isLineBuilt = true and
        id not in set dom trains and
        moduleId in set dom modules and
        modules(moduleId).getIsStation() and -- We can only put new trains in stations
        not exists t1 in set dom trains & trains(t1).getCurrentBlock().getModule() =
modules(moduleId) -- The station must be empty
        post id in set dom trains and
         trains(id).getCurrentBlock().getSemaphore(orientation).getColor() = <Red> and
trains(id).getCurrentBlock().getSemaphore(orientation).getSensor() = <Busy> and -- The
block is occupied and busy
                trains(id).getCurrentBlock().getModule() = modules(moduleId);
```

Neste caso, o uso do *operador* ++ em:

```
trains := trains ++ {id |-> new Train(id, orientation,
modules(moduleId).getBlock(orientation))};
```

garante que o *map* "trains" é acedido dentro do seu domínio em qualquer situação, dado que o parâmetro *id* não é modificado no corpo desta operação.

5.2 - Exemplo de verificação de uma invariante

Número	Nome da *Proof Obligation*	Tipo
49	*TrainLine`addModule(String, Module)*	*State invariant holds*

O código em análise é o seguinte (com o *assignment* a vermelho):

```
        -- Adds a module to the station
        private addModule: (String) * (Module) ==> ()
        addModule(newModuleId, newModule) ==
        (
                if card dom modules > 0 -- if we already have a module added
                then
                (
                        dcl tailModule: Module := modules(tailModuleId);

                        -- Connect the new model to the last model insered:
                        tailModule.setUpModule(newModule);
                        newModule.setDownModule(tailModule);

                        -- Update the last model inserted:
                        tailModuleId := newModuleId;
```

```
            )
            else
            (
                  atomic
                  (
                        headModuleId := newModuleId;
                        tailModuleId := newModuleId;
                  );
            );

            modules := modules munion { newModuleId |-> newModule };
      )
      pre isLineBuilt = false and newModuleId not in set dom modules
      post newModuleId in set dom modules and -- The id was added to the modules
        if card dom modules~ = 0
            then (headModuleId = newModuleId and tailModuleId = newModuleId)
            else (tailModuleId = newModuleId and
modules(tailModuleId~).getUpModule() = newModule and newModule.getDownModule() =
modules(tailModuleId~));
```

As invariantes na classe *TrainLine* são as seguintes:

```
-- headModuleId and tailModuleId must be both null before any modules being added
inv (headModuleId = nil and tailModuleId = nil) => card dom modules = 0;
inv isLineBuilt = true => (headModuleId <> nil and tailModuleId <> nil);

-- There arent any trains in the same block
inv isLineBuilt => not exists t1, t2 in set dom trains & (t1 <> t2 and
trains(t1).getCurrentBlock() = trains(t2).getCurrentBlock());

-- There must be a minimum of 2 stations
inv isLineBuilt => exists s1, s2 in set dom modules & (s1 <> s2 and
modules(s1).getIsStation() and modules(s2).getIsStation());

-- There must be at least 1 module between stations and at least 2 stations
inv isLineBuilt => forall m in set dom modules &
            (modules(m).getIsStation() =>
                  modules(m).getNextBlock(<Up>).isInStation() = false
                  and modules(m).getNextBlock(<Down>).isInStation() = false)
```

As últimas quatro invariantes nunca serão quebradas, porque a pré-condição do método onde se encontra o *assignment* obriga a que *isLineBuilt* seja falso. No caso da primeira invariante, a porção de código onde se encontra o *assignment* só é corrido caso exista pelo menos um tuplo no *map*, e quando tal acontece, *headModuleId* e *tailModuleId* não são *nil*, logo a parte da esquerda da invariante nunca será *true*.

6 - Geração de Código-fonte Java

O código fonte *Java* foi gerado a partir do *Overture*. De seguida, foram adicionadas novas classes que utilizam as classes geradas do modelo de *VDM++*, de modo a criar uma aplicação de linha de comandos.

A classe que contém a função *main* da aplicação é a *CommandLineInterface*, contida no package *cli*.

A aplicação começa por mostrar o menu principal (*Figura 11*), onde é possível escolher opções para:

- *Visualizar o mapa da linha férrea*
- *Visualizar a lista dos comboios existentes*
- *Visualizar o estado de um comboio*
- *Visualizar o estado de um semáforo/sensor.*
- *Visualizar o estado de um bloco*
- *Selecionar um comboio*
- *Ligar ou desligar um semáforo*
- *Ligar ou desligar um sensor*

Para selecionar uma entrada do menu, é necessário introduzir o número correspondente à opção. Para selecionar um objeto, é necessário introduzir o identificador correspondente.

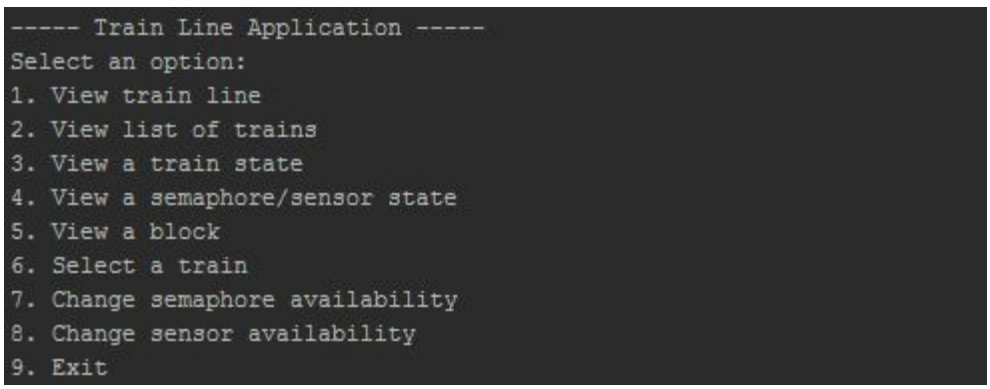

Fig. 11 - Menu principal da aplicação.

A opção *selecionar um comboio* permite aceder a um novo menu que permite controlar o comboio escolhido (*Figura 12*).

Fig. 12- Menu de controlo do comboio.

A partir deste menu, é possível ver o estado atual, que inclui o bloco em que o comboio se encontra e a informação do estado do próximo semáforo. É ainda possível mover e parar o comboio e pedir permissão para sair de uma estação.

7 - Conclusões

Relativamente ao trabalho desenvolvido, o grupo cumpriu todos os objetivos propostos na especificação, tendo sido obtido um modelo que consegue evitar totalmente as colisões entre os comboios. No entanto, o modelo é um pouco restritivo no que toca à aceitação de um pedido de um comboio para sair de uma estação.

Relativamente aos aspetos a melhorar, poderia ser desenvolvido um algoritmo mais flexível e que, ao mesmo tempo, assegurasse que não houvessem colisões ou situações de congestionamento entre os comboios.

Os dois elementos do grupo distribuíram o trabalho equitativamente entre si.

8 - Referências

[1] *Rail Directions*. URL: https://en.wikipedia.org/wiki/Rail_directions (acedido em 14 de dezembro de 2015).