On the Use of Coloured Petri Nets for Visual Animation [⋆]

Óscar R. Ribeiro João M. Fernandes

Dep. Informática / CCTC, Universidade do Minho, Braga, Portugal

Abstract. This paper reports on an exercise on constructing a visual animation layer for a behaviourally-intensive reactive system. We assume that the requirements of the system under consideration are described by use cases, and the behaviour of each use case is detailed by a collection of scenario descriptions. These use cases and scenarios are translated into a Coloured Petri Net (CPN) model, which is subsequently complemented with animation-specific elements. We describe how the CPN model must be structured to facilitate the animation process, and we present the supporting tools for creating the animation. We consider an elevator controller system as a case study, to demonstrate that a CPN model complemented with a visual animation layer constitutes a solid basis for addressing behavioural issues in an early phase of the development process, namely during the validation task.

1 Introduction

Validation consists on checking if a system or a model satisfies the user expectations. One of the key issues to have a successful validation is to adopt a process where users can actively discuss the requirements of the system under development. To accomplish the validation task, one can use a visual animation [1] that employs domain-specific vocabulary, since it allows users and developers to be confident that the right system is being built.

When developing a reactive system [2], which usually has an intensive behaviour and a rich set of interactions with its environment, requirements validation is an important task to accomplish before deciding any design and implementation issues.

The Unified Modeling Language (UML) is currently the standard notation adopted in industry to model software systems. This work uses two UML diagrams: Use Case Diagrams and Sequence Diagrams. Use cases specify the set of functionalities presented by a system as seen by its users, and permit, due to their simplicity, the dialogue between clients and developers. A sequence diagram is used to capture a behavioural scenario of a given system, which can be seen as a sequence of steps describing interactions between the actors and that system. We suggest each use case to be described by a set of sequence diagrams.

[⋆] This work has been supported by the grant with reference SFRH/BD/19718/2004 from "Fundação para a Ciência e Tecnologia".

Coloured Petri Net (CPNs) [3] constitute a graphical modelling language appropriate to describe the behaviour of systems with characteristics like concurrency, resource sharing, and synchronization. The CPN Tools [4, 5] is a well established tool supporting the CPN modelling language and allowing the execution of animations in accordance with the CPN model.

This paper describes the construction of a visual animation layer for the problem domain of the elevator controller case study. The visual animation can be used during the validation to facilitate the dialogue between the system developers and the clients. This animation layer described in this work is intended to be controlled by an executable CPN model. We give some guidelines to create the CPN model from the sequence diagrams present in the requirements of the system under development. We also introduce a tool to support the development of the animation layer.

This paper is structured as follows. Section 2 describes the elevator controller case study to be considered in this paper. In Section 3, we present the development of an animation layer for the elevator controller. We introduce also some tools to support this development. Section 4 introduces some guidelines to construct a CPN model for animation purposes. The conclusions are presented in Section 5.

2 Case Study

In this section we introduce the case study used in this paper, an elevator controller. We consider that this controller manages an elevator system with two cars in a building with six floors. This case study is an adaptation from the description presented in the technical report [6].

Fig. 1 depicts the context diagram for the elevator controller, where the main sensors and actuators present in the considered top-most entities (Floor and Car) are shown. Each Floor contains two Location Sensors, one for each car, to detect when the respective car is at or is arriving to the floor. In each Floor there are Hall Buttons to allow the passengers to call an elevator car indicating the direction (up or down) he wishes to travel. Obviously, the first and the last floors have only one button to select the unique direction that is possible to travel (up for the first floor and down for the top-most floor). There is a Floor Door in each floor to protect the car's shaft, and the doors only open when the corresponding car is stopped at the floor.

Each of the two Cars has one Car Motor to move up or down along the floors. Each car has a Door which has: (1) a Door Timer to automatically close the door after a given amount of time; (2) a Door Sensor to detect if there is something obstructing the door during closure; (3) a Door Motor to open/close the door of the elevator; and (4) a Car Door, which includes two sensors, to indicate either if the door is closed or totally opened. The Car Door is mechanically linked with the Door Motor, and also with the corresponding door in each floor.

Inside the car a passenger can find a Floor Indicator that shows the current floor of the car, a Direction Indicator that shows the direction being followed by

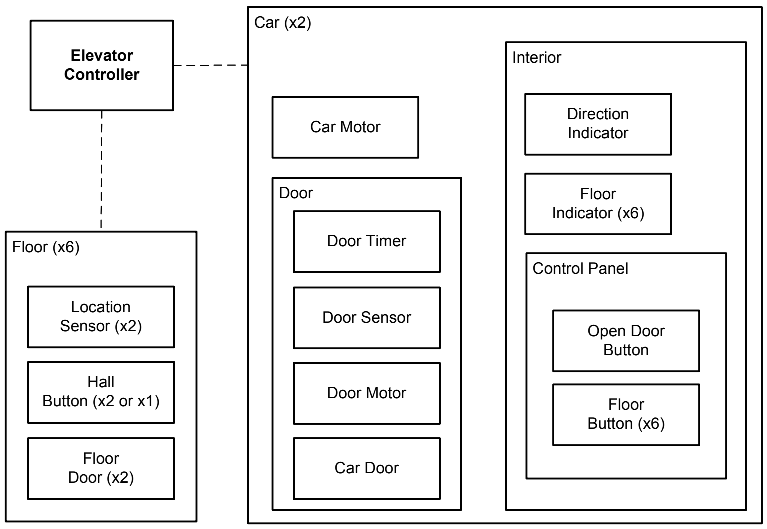

Fig. 1. Context Diagram for the Elevator Controller

the car, and a Control Panel that contains an Open Door Button to open the doors and six Floor Buttons to select the destination floor.

The behaviour of the elevator controller is triggered by the passengers' actions. The elevator controller has the responsibility of managing the movements of the cars in accordance to the requests of the passengers through pushing on the buttons.

Our approach proposes the usage of a use case diagram to depict the main functionalities provided by the system to its users. In this paper, with the purpose of illustrating the suggested approach, we just consider the use case "Service Floor". This use case is responsible for moving an elevator car from an origin floor to a destination floor, by request of a passenger either in one of the building's hall or inside an elevator car.

Fig. 2 depicts the sequence diagram with the main scenario of the "Service Floor" use case. This sequence diagram uses some high-level operators present in the UML 2.0, namely the opt, and the loop operators. These operators permit the description of several scenarios in a unique sequence diagram. In order to abstract from the scenarios presented in the sequence diagram, there are along the left-hand side of the diagram some textual annotations where some variables being used in the messages (and guards) are informally declared. For example, the diagram in Fig. 2 abstracts from the car, and the origin and destination floors. With the textual annotation *"Elevator car c is at Floor Fo, and the next requested floor is the Fd"*, we are declaring the variables c, F_o (origin floor) and

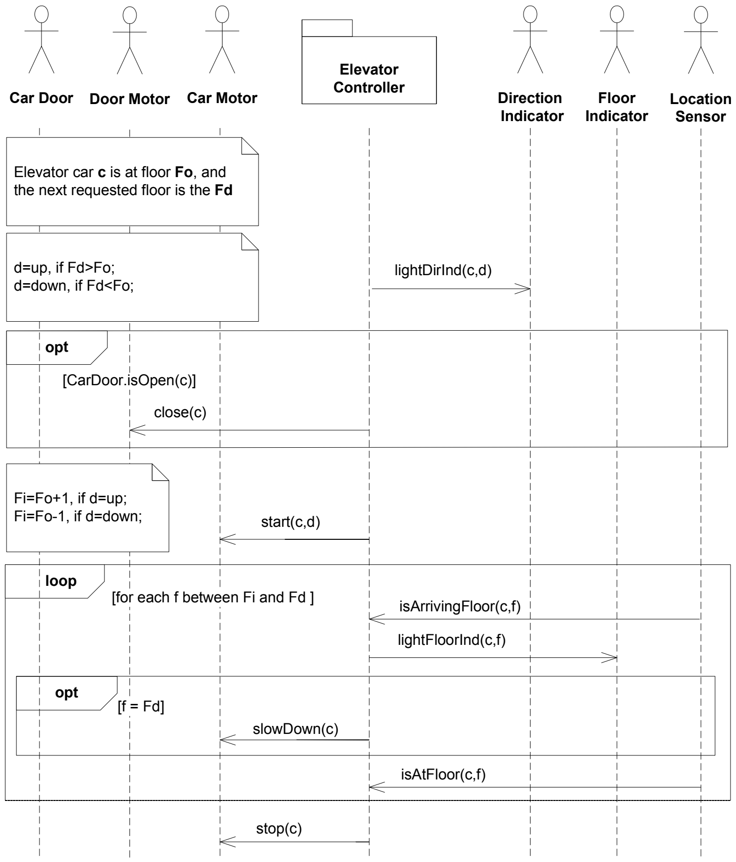

Fig. 2. Sequence Diagram describing the "Service Floor" use case.

F_d (destination floor), to be used as parameters for the scenario present in the diagram. With these variables, we are able to define variable d that represents the direction followed by the car, in the textual annotation "d=up, if Fd>Fo; d=down, if Fd<Fo". The textual annotation "Fi=Fo+1, if d=up; Fi=Fo-1, if d=down;" defines the variable F_i, representing the next floor from the origin floor.

The scenario presented in Fig. 2 describes the following behaviour:

1. The passenger in the current floor is notified about the direction the car will take (message lightDirInd);
2. If the car door is open (high-level operator opt)
   (a) The car door is closed (message close)
3. The car is moved in direction to the destination floor (message start);
4. While the destination floor is not reached (high-level operator loop):

(a) The location sensor informs the elevator controller that the car is arriving to the next floor (message isArrivingFloor);
(b) The Floor Indicator corresponding to the next floor is activated (message lightFloorInd);
(c) The Car Door must slow down its speed, when the next floor is the destination floor (message slowDown);
(d) The location sensor informs the elevator controller that the car is at the next floor (message isAtFloor);

5. The car stops (message stop).

In this main scenario of the "Service Floor" use case, we are assuming that during its execution there are no other interaction from the passengers with the buttons. These situations could origin some other variations to this main scenario, but in this work we will consider only the main scenario of the use case.

3 Building an Animation to the Case Study

In this section, we present the development of an animation layer for the elevator controller introduced in the previous section. We describe also how to use the animation, and which tools are involved in its deployment.

3.1 Initial Considerations

The visual animation to be built is intended to reproduce the behaviour of the elevator controller, specified by the considered collection of sequence diagrams. It can be used during the validation to facilitate the dialogue between the system developers and the clients, which is a critical facility to ensure that both parts agree on what is to be developed.

One can start the construction of a visual animation, after some initial work on the analysis task has been accomplished. For this purpose, it is useful to have the context diagram, since it enumerates the main entities of the environment with which the controller interacts. Typically, these entities are strong candidates to be represented in the animation, since the system's behaviour depends on them.

It is also important to have the use case descriptions, together with their corresponding sequence diagrams, since they describe which messages are received by the elements in the environment, and how these elements react on those messages. These artefacts specify the behaviour that must be covered by the animation layer.

The construction of the animation must be synchronised with the activity of creating the CPN model, because the CPN model must include some elements which are specific to control the animation as explained in next section.

The animation has been created using the SceneBeans tool [7, 8], which is a framework for creating and controlling animations, using the Java programming language. There is a XML-based file format to define animations and a parser to translate those XML files into animation objects for SceneBeans.

In the SceneBeans architecture there are as basic elements scene graphs, behaviours and animations. A *scene graph* is implemented in JavaBeans by a direct acyclic graph, which draws a two-dimensional image. In the leaf nodes of a scene graph there are primitive shapes (such as circles, ellipses, rectangles). An intermediate node either combines or transforms its subgraphs. The combination can be done in two ways: putting one subgraph on the top of another; or choosing one from the set of subgraphs. In a transform node it is possible to apply a linear transformation followed by a translation to its subgraphs (for example rotation, scaling or translation) or to change the way that its subgraphs are rendered (for example, changing the colour in which a node is drawn).

Associated to each graphic element in the animation, there are some behaviours (not to be confused with the behaviour of the controller) to animate some of the properties of the element. A *behaviour* in SceneBeans is implemented by a Java bean that controls a time value, and when the value changes it announces an event. This permits the animation of the visual appearance of the scene graph. Notice that there is a so-called *animation* thread that manages the frame rate of the overall animation signals the passage of time. There is also the possibility to define commands to call the execution of a set of behaviours and to announce an event when they finish.

3.2 Static part of the animation

The behaviour of the elevator controller, as the behaviour of reactive systems in general, lies in the interaction with its environment, by sending messages to the environment, which in response can also send messages in the opposite direction (i.e., to the controller). In this work we consider the elements in the environment as the actors of the elevator controller system. The actors for the "Service Floor" use case are the ones that participate in the sequence diagram in Fig. 2.

An elevator can be visually represented by a picture with the floors and the cars, where the cars can go up/down across the floors. While the cars are moving, the elevator controller must update the information shown in the panels inside each elevator car and attend the requests from the passengers.

For the elevator controller, only a subset of the entities of the environment are relevant for animation, since the passenger is not aware of (or does not interact with) all of them. This means that the animation layer only includes the relevant entities. In contrast, the CPN model does specify all the environment's entities. In the elevator controller, the passenger interacts only with the following six entities: Hall Button, Car Door, Direction Indicator, Floor Indicator, Open Door Button, and the Floor Button. The validation focuses essentially on the reactions of the controller to requests made by the passengers. If some flaw on the behaviour of the elevator is detected during validation, the developer may also need to analyse all the entities of the environment (even those that do not appear in the visual animation), to identify and understand the cause of the error.

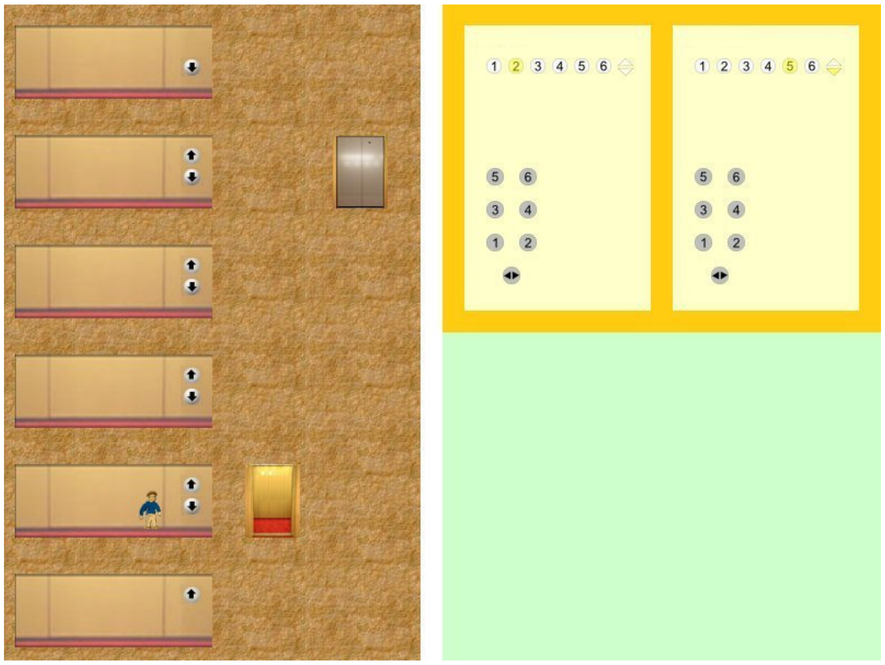

Fig. 3. An animation of the elevator system.

Fig. 3 shows a screenshot obtained from the animation of the elevator controller. On the left part of the figure, there is a representation of each floor with the buttons to call an elevator. We can also see the two elevator cars in Fig. 3: the left-hand side elevator is at the second floor with its door open, and the right-hand side elevator is at the fifth floor with its door closed.

On the right part of the figure, the interior of both cars are depicted. Each car has a floor indicator, which has one light for each floor, and only the light of the current floor is on. There are also two lights, one to indicate the up direction and another one to indicate the down direction, showing the direction being followed by the car. Fig. 3 shows that the left-hand side car has the light with the number two in yellow indicating that this light is on, and thus that the car is currently at the second floor. The left-hand side car currently has no direction, and the light direction of the right-hand side car indicates that the car is going down.

The structure of the animation is essentially based on importing some icons to represent an element of the system or on drawing a geometric artefact using a XML tag. This structure is present in the leaf nodes of the scene graph, and for example to include the image door.png we use the XML tag primitive, as follows:

```
<primitive type="sprite">
   <param name="src" value="./images/door.png"/>
</primitive>
```

3.3 Dynamic part of the animation

In this subsection we show how to define the dynamic part of an animation in the SceneBeans XML-based format.

As we said before, in the leaf nodes of a scene graph there are primitive shapes and in the intermediate nodes either combination or transformation of its subgraphs using a set of parameters. Each parameter can be associated to a behaviour that needs to be previously defined. To allow the user to control the existing behaviours, there are commands, and each one includes a sequence of behaviour invocations. Thus, calling a command results on the animation of some of the parameters present in the intermediate nodes.

With respect to the animation of our case study, let us consider how to specify in the XML-based format the animation of the Direction Indicator entity and, in particular, how the message lightDirInd is handled. This message allows the Direction Indicator to change its state, among its possible values (up, down, and idle). The Direction Indicator is composed of two triangular lights. If the car is going up (down), the top light "△" is on (off) and the bottom light "▽" is off (on). If the car is stopped, the idle direction is represented by delighting both lights.

For example, to indicate that the car is going up, we use the following XML code, whose XML tags param in lines 2-4 are used to set the parameters from, to and duration of the behaviour:

```
1 <behaviour algorithm="move" event="ldi" id="showlightDirInd(rightCar,up)">
2    <param name="from" value="-1000"/>
3    <param name="to" value="750"/>
4    <param name="duration" value="0.0001"/>
5 </behaviour>
```

This block of code defines a behaviour that is based on a specific movement of an animation icon from a given point ('from') to another point ('to') during a given time ('duration'). This behaviour will be associated to the parameters in the nodes of the scene graph.

To indicate, during the animation, that the car is going up, it is needed to put an icon (showing the top light on, and the bottom light off) at the position of the Direction Indicator entity in the animation picture. This is achieved by the following XML code:

```
<transform type="translate">
      <param name="translation" value="(-1000,50)"/>
      <animate param="x" behaviour="showlightDirInd(rightCar,up)"/>
      <animate param="x" behaviour="hidelightDirInd(rightCar,up)"/>
         <primitive type="sprite">
            <param name="src" value="./images/direction_indicator_up.png"/>
         </primitive>
</transform>
```

In lines 3 and 4, behaviours showlightDirInd (presented in the previous block of code) and hidelightDirInd are associated to the parameter "x" of the transformation node, in order to change the x-axis position of the icon, i.e., they move the icon horizontally in the animation picture. These behaviours are invoked through their inclusion in a command definition as we present in the next block of code.

There is one icon to represent each state of the Direction Indicator entity, and showlightDirInd behaviour moves the respective icon to a visible part of the animation, and the hidelightDirInd behaviour moves the respective icon to a non-visible part of the animation. Thus, the change to a new state is animated showing the icon of the new state and hiding the other two icons.

To allow the external invocation of these behaviours in order to animate the changing of the direction indicator in the car on the right-hand side to indicate the up direction, the following command announces an event with the same name.

```
<command name="lightDirInd(rightCar,up)">
   <start behaviour="hidelightDirInd(rightCar,down)"/>
   <start behaviour="hidelightDirInd(rightCar,idle)"/>
   <start behaviour="showlightDirInd(rightCar,up)"/>
   <announce event="lightDirInd(rightCar,up)"/>
</command>
```

There are similar code blocks for the other two possible directions and for the other car.

3.4 Scripting Language

We have created a script to facilitate the manipulation of the XML tags in the XML-file that specify the animation. Ruby [9, 10] is a scripting language that follows the principles of object-oriented programming. The Ruby script uses components from the library REXML [11].

With Ruby we can easily manipulate the XML to be generated, namely when repetitive parts of the XML code follow a given pattern. To generate the XML code for the animation, a Ruby script was created. Next we show part of that script that generates the XML code for the animation of the Direction Indicator.

```
require 'builder'
require 'sbXMLgen.rb'

xmlBuilder = Builder::XmlMarkup.new(:target => $stdout, :indent => 3)

xmlBuilder.instruct! :xml, :version => "1.0"
xmlBuilder.animation("width" =>"800", "height" => "600" ){
  carIds = ["left","right"]
  carInteriorCoord = {"left" => Tuple.new(420,20), "right" => Tuple.new(610,20)}
  lstBhLightDI = Hash.new()

  lstCommands = Hash.new()
  carIds.each{ |carId|
     bhparams = Hash.new()
     durQuick = 0.0001
     xHide = -1000
14      xVisible =   carInteriorCoord[carId].getX+7*20
15     bhparams["show"] = BehaviourParams.new(xHide,xVisible,durQuick)
16       bhparams["hide"] = BehaviourParams.new(xVisible,xHide,durQuick)
17      directions = ["up", "down", "idle"]
18      movs = ["show", "hide"]
19      bh = Hash.new()
20      movs.each{ |m| bh[m] = Hash.new() }
21      lstBhLightDI[carId] =  {"up"=>[], "down"=>[], "idle"=>[]}
22      mkCmdStrDI= lambda{|mov, cId, dir| "#{mov}lightDirInd(#{cId}Car,#{dir})"}

23      directions.each{ |d|
24          movs.each{ |mov|
25            cmdStrDI = mkCmdStrDI.call(mov,carId,d)
26             bh[mov][d] = Behaviour.new(cmdStrDI, "move", bhparams[mov], "xpto")
27             bh[mov][d].toXML(xmlBuilder)
28            lstBhLightDI[carId][d].push(Tuple.new("x", cmdStrDI))
29            if (mov=="show") then
30                d_tmp = directions.dup
31                d_tmp.delete(d)
32                lstBhToStart = Array.new()
33                 lstBhToStart.push(cmdStrDI)
34                d_tmp.each{ |od|
35                   cmdStrDIod = mkCmdStrDI.call("hide",carId,od)
36                   lstBhToStart.push(cmdStrDIod)
37                }
38                cmdStrDIname = mkCmdStrDI.call("",carId,d)
39                lstCommands[cmdStrDIname] = lstBhToStart.dup

40            end
41          } }
42  }
43  lstCommands.each{ |k,v|   make_command(xmlBuilder, k , v) }
44   xmlBuilder.draw {
45    carIds.each{ |carId|
46       draw_DirectionIndicator(xmlBuilder, carInteriorCoord[carId],
47                                                    lstBhLightDI[carId])}
48 }
```

In line 1, the package from the REXML library to build XML tags is included. When we define an object as a Builder (see line 3 where we define the variable xmlBuilder) we are able to generate a XML tag by using the name of the tag to be generated as as a method for the object, for example in line 5 is created the following XML structure:

```
<animation height="600" width="800">
...
</animation>
```

Inside the tag animation it is included the XML code generated by the lines 6-47 of the Ruby script.

In line 6, the variable carIds is defined as an array with the car identifiers. It is possible to create an iteration over this variable, as is shown in line 10. This permits an easy integration of new similar cars in the animation, because we have a dynamic structure based on the elements into the array carIds.

Line 2 includes some code from file “sbXMLgen.rb” created by us, in order to save some functions to be used on the generation of XML for animations, independently from the case being considered. For example, the code in line 26 creates an object of the class Behaviour which has the following definition:

```
class Behaviour
   def initialize(id, algorithm, params,  event, announce = "no")
      @id     = id
    @algorithm = algorithm
      @params = params  # class BehaviourParams
    @event = event
      @toBeAnnounced = announce
  end
   def toXML(xb)
    xb.behaviour("id" =>@id, "algorithm"=> @algorithm , "event" => @event){
         @params.toXML(xb)
            }
   end
end
```

This a simple definition of a Ruby class, where we can find the same parameters as the behaviour XML tag, and the definition of the method toXML to generate the corresponding XML code. The advantage is that we can use this class to have a less verbose (i.e., easier to read by humans) code to specify a behaviour. Consequently, if we repeat this process for all other XML tags present in the SceneBeans XML-based format, we obtain a less verbose way to define an animation layer.

We believe that this Ruby Script constitutes an abstract way to deal with the XML-based stuff, in particular it is an easy way to work with parameterising in the visualizations. When comparing it with the forall construct we think that the Ruby script is a more flexible and user-friendly way to manipulate the parameters. The forall construct has the advantage that it is defined in the same XML-file as the rest of the animation definitions.

4 Coloured Petri Net Models

In this section, we describe some guidelines that we suggest to be taken into account when creating a CPN model for animation purposes. These guidelines are discussed and are exemplified with respect to the case study.

The CPN model that is constructed from a set of scenario descriptions is an executable model that can drive a graphical animation layer showing elements and concepts from the problem domain. Additionally, the CPN model needs to include a mechanism to manage how animation events are handled. The animation layer, in the SceneBeans XML-based format for the case study considered in this work, is presented in the previous section.

The BRITNeY suite [12, 13] animation tool is used to connect the execution of the CPN model in the CPN tools with the SceneBeans objects corresponding to the animation specified in XML-based file. We use the SceneBeans plug-in present in the BRITNeY suite to display and interact with a SceneBeans animation.

As stated before, we consider that the requirements process includes the creation of a set of use cases. The behaviour of each use case is detailed by a collection of scenario descriptions, which can be represented by sequence diagrams. In version 2.0 of UML, sequence diagrams have many high-level flow operators.

The translation from these sequence diagrams to a CPN model is based on the general principles described in [14], which associate a transition on the CPN model for each message in the sequence diagram and define some mechanisms in the CPN model to represent the high-level operators present in the sequence diagrams.

We describe the construction of a CPN model to execute the scenarios described by the sequence diagram in Fig. 2, and also the changes that need to be accomplished to allow the obtained CPN model to regulate the animation introduced in the previous section.

4.1 Mapping sequence diagrams into a CPN model

Firstly, we suggest that each sequence diagram is translated to a CPN model where there is a substitution transition for each message or high-level operator in the sequence diagram. The places between substitution transitions guarantee the order between the messages in the sequence diagram, and their colour set needs to include the necessary information to allow the parallel execution of many scenarios.

Fig. 4 shows a CPN model that was obtained from the sequence diagram in Fig. 2, where messages, and high-level operators in the sequence diagram are represented by substitution transitions in the CPN model. For example, the message lightDirInd is represented by the substitution transition with the same name, and the first opt operator in the sequence diagram is represented by the substitution transition "opt (Is car door open?)". The messages inside the opt operator are present inside the corresponding subpage. The subpages contain the necessary details to animate the messages in the sequence diagram.

Places in the CPN of Fig. 4 have the colour set ScenarioUC2, which provides the necessary information to execute a scenario of "Service Floor" use case, namely the origin and destination floors and the car being used. In other words, the definition of the colour set ScenarioUC2 comes from the textual annotation *"Elevator car c is at floor Fo, and the next requested floor is the Fd"*, from the sequence diagram in Fig. 2, where the variables c, F_o and F_d are informally declared. These implicit and informal declarations of variables by a textual annotation allow for the usage of the same sequence diagram in different situations, by using the variables as a parameter in messages, or even in other textual annotations. The definition of the colour set ScenarioUC2 in the CPN ML programming language has the following code:

```
colset ScenarioUC2tmp = record
                       c:  CarId *
                       fo: FloorNumber *
                       fd: FloorNumber ;

fun hasDiffFloors(s:ScenarioUC2tmp)= (#fo s) <> (#fd s) ;

colset ScenarioUC2 = subset ScenarioUC2tmp by hasDiffFloors;
```

CarId and FloorNumber are defined as integers to identify a car and a floor, respectively. The colour set ScenarioUC2tmp is created essentially to be used in

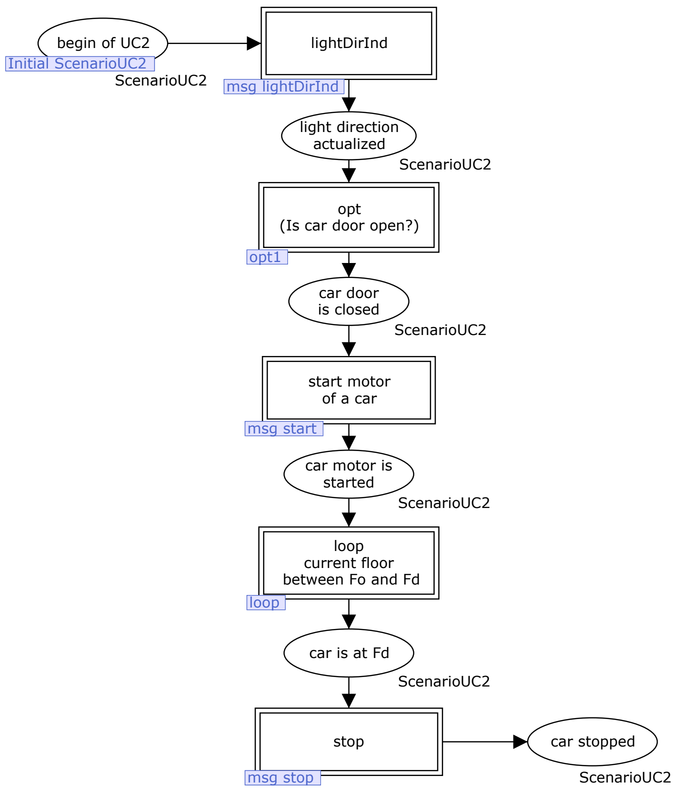

Fig. 4. CPN model representing the sequence diagram of UC2.

the definition of the colour set ScenarioUC2. When executing an instance of the "Service Floor" use case, it is implicit that the origin and destination floors are different. This is specified using the predicate hasDiffFloors to restrict the colour set ScenarioUC2tmp to obtain the colour set ScenarioUC2, whose elements are guaranteed to have different origin and destination floors.

The colour set ScenarioUC2 is used to distinguish between parallel executions of the use case, and thus the colour set must identify the car, the origin floor, and the destination floor. To start the execution of the CPN model it is necessary to define the input parameters using the place begin of UC2, where several tokens can be put to allow the parallel execution of different UC2 instances.

The places in the CPN model of Fig. 4 ensures that the order between messages in the sequence diagram is maintained when firing the transitions in the CPN model. Each token in the place begin of UC2 means that a "Service Floor" has been requested to be executed, i.e., a given car must travel from a origin to a destination floor. It is assumed that the environment is in conditions to allow the execution of this scenario, provided by other use cases (not considered in this paper). The conditions to permit the execution of an instance of the "Service Floor" use case are that the selected car must be at the origin floor, with its motor stopped.

A token in the place light direction actualized means that the direction indicator light is now indicating the direction that the car is taking to go from the origin to the destination floor, and allows the next transition to be enabled.

4.2 Data representation for the environment

Secondly, it is important to define a data representation of the main elements in the environment of the system under development. This data description is used to represent the behaviour of each message in the sequence diagram.

To obtain a description of the system's environment, its elements are specified as data in the CPN model using the CPN ML programming language, defining a colour set for each element in the environment. In our case study, the cars and the floors are the top-most entities of the environment. For example, the colour set Car is a record with an identification of the car, the door of the car, the motor to move the car, and the sensors and actuators inside the car.

```
colset Car = record
               id        : CarId       *
               motor     : CarMotor    *
               door      : CarDoor     *
               interior  : CarInterior ;
```

Where the CarId colour set is an integer, the CarMotor colour set is a tuple containing its moving speed and the direction being followed. The CarDoor colour set is defined as a record stating if the door is open or closed, and representing also the motor, the sensor and the timer of the door.

```
colset CarDoor = record
                    door      : Door      *
                    doorMotor : DoorMotor *
                    doorSensor: DoorSensor*
                    doorTimer : DoorTimer ;
```

Similarly, the entities inside a car are defined by the colour set CarInterior and they include the lights to indicate the direction being followed by the car, the lights to indicate the current floor (there is a light for each floor), and a control panel where we can find a button to open the door, and buttons to allow the passenger to select one of the existing floors.

```
colset CarInterior = record
                       directionIndicator : Direction   *
                       floorIndicator     : FloorNumber *
                       controlPanel       : ControlPanel;
```

We consider that the components in the car are part of the colour set, such as the car door, the door motor, the door sensor, the door timer and the car motor.

The textual annotation "$d = up,\ if\ Fd > Fo;\ d = down,\ if\ Fd < Fo;$" in the sequence diagram of Fig. 2 is represented by the function calcDirection which takes a ScenarioUC2 colour set and gives a direction based on the origin and destination floors, and is defined as follows:

```
colset Direction = with up | down | idle ;

fun calcDirection(a:ScenarioUC2) =
    case (Int.compare(#fo p,#fd p)) of
       LESS    => up
      |GREATER => down
      |EQUAL   => idle ;
```

Some of the messages in the sequence diagram have the parameter direction, which can be calculated using the values of the ScenarioUC2. For example, the message lightDirInd uses the parameter direction.

4.3 Animation of messages in the sequence diagrams

Thirdly, one must detail how the execution of each substitute transition in the CPN model representing a message in the sequence diagram is animated in the SceneBeans animation.

The communication between the CPN model and the SceneBeans animation is done using an animation object which can be used in the code segments of CPN model to invoke some commands to be executed in the SceneBeans animation. The CPN model contains the following declaration of the "anim" as a SceneBeans object:

```
structure anim = SceneBeans(val name = "Elevator Controller Animation");
```

Fig. 5 shows the subpage associated to the message lightDirInd. The animation of a message in the sequence diagram is divided in two transitions of the CPN model: the first one (lightDirInd) to invoke the command in the animation, and the second one (ack lightDirInd) to wait for the feedback from the animation, informing that the command has been executed. Thus, a token in the place waiting for lightDirInd event means that the animation is updating the appearance of the direction indicator in the animation. This mechanism, that waits for the event from the animation, ensures the synchronization between the animation and the execution of the CPN model.

To ask for an execution of a command in the SceneBeans animation we use the "invokeCommand" method which can be used for an object with an SceneBeans animation. The parameter for this method is a string which must correspond to a command in the animation. The command lightDirInd is used to animate the Direction Indicator, as defined in the previous section, and has two parameters: (1) the name of the car, and (2) the direction that the car is taking. To ease the creation of the command identifier, the following function can be created:

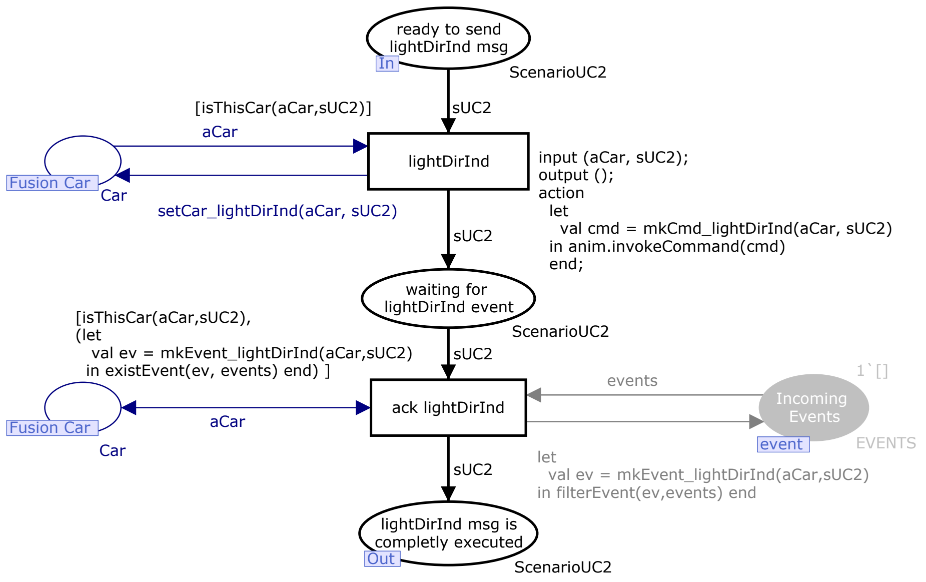
ready to send lightDirInd msg
In
ScenarioUC2
sUC2
[isThisCar(aCar,sUC2)]
aCar
lightDirInd
Fusion Car
Car
setCar_lightDirInd(aCar, sUC2)
input (aCar, sUC2);
output ();
action
let
val cmd = mkCmd_lightDirInd(aCar, sUC2)
in anim.invokeCommand(cmd)
end;
sUC2
waiting for lightDirInd event
ScenarioUC2
[isThisCar(aCar,sUC2),
(let
val ev = mkEvent_lightDirInd(aCar,sUC2)
in existEvent(ev, events) end)]
sUC2
events
1`[]
Incoming Events
Fusion Car
aCar
ack lightDirInd
event
EVENTS
Car
let
val ev = mkEvent_lightDirInd(aCar,sUC2)
in filterEvent(ev,events) end
sUC2
lightDirInd msg is completly executed
Out
ScenarioUC2

Fig. 5. CPN model for the execution of the message lightDirInd.

```
fun mkCmd_lightDirInd(aCar:Car, sUC2: ScenarioUC2) =
let
    val sc     = carName(aCar)
    val  strd = showDir(calcDirection(#floors sUC2))
in "lightDirInd("^sc ^"Car,"^ strd ^")" end
```

To verify the incoming of the corresponding event of the command lightDirInd it is necessary to test if the event is in the list of incoming events. Our CPN model is constantly aware of the events being generated by the animation through the module in Fig. 6. The running fusion place is used to test if the animation is already started.

4.4 Initial conditions for scenario execution

Fourthly, we suggest to add a module to the CPN model where the initial conditions for the scenario execution and the selection of the SceneBeans XML file to used are introduced.

The CPN model in Fig. 7 analyses the initial conditions of the environment that the user wants to simulate and subsequently initialises the animation. The initial conditions are introduced in the pre-places (in green) of transition Initialise. The firing of the Initialise transition invokes the necessary commands to start running the animation according to the specified conditions. The Fig. 7 constitutes a top page of the CPN model where for each scenario a separated

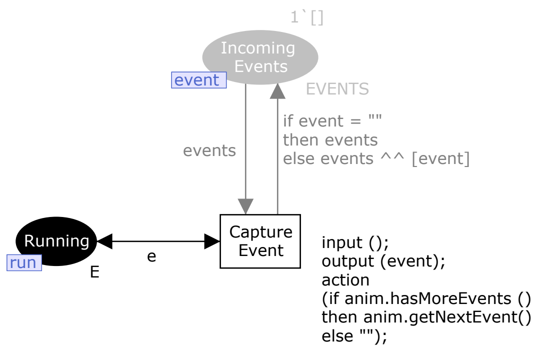

Fig. 6. Subpage of CPN model to capture events from the SceneBeans animation.

subpage exists and for which the start place can be found in Fig. 7 thought a fusion place. The running place is used to allow the execution of CPN model in subpage in Fig.6. There are also fusion places to connect the places with the initial values with places that represent the environment values in the subpages.

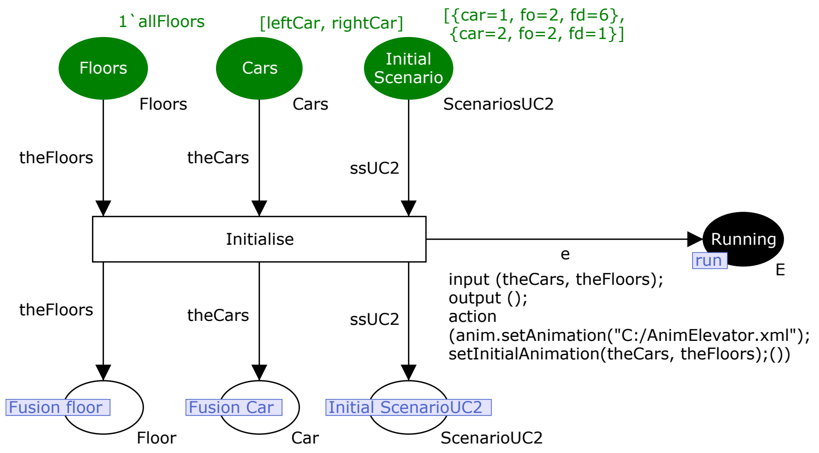

Fig. 7. CPN-model to initialise the environment values and the SceneBeans animation.

Let us see now how to change the CPN model to allow the consideration of a different numbers of cars and floors in the animation. To do this we need to adapt both the CPN model and the ruby file presented in page 9.

The changes in the CPN model are done in the topmost page presented in Fig. 7, changing the initial marking of the pre-places for Initialise transition. It is also necessary to change the value of a constant that represents the number of floors and another that represents the number of cars. These changes must be complemented with the corresponding changes in the Ruby file namely in what concerns with the variables that represent the available cars and the available floors. And also the position of different elements in the graphic animation.

Supposing that we want to introduce a new car in the middle of the other two existing cars we define a constant, e.g. centerCar, with its data representation, to be included in the list of initial marking in place "Cars" of Fig. 7, which results in the list [leftCar, centerCar, rightCar]. If we want to execute a scenario related to this new car we need to add the scenario description to the list of initial scenarios to be executed. Now we will refer to the Ruby script code listed in page 9. Firstly in line 5, it is necessary to increase the height of the animation to have some space to introduce the icon for the additional car. In line 6 we add the identification label that we want to use for the identification of the new car, and in line 7 we add to the hash table the coordinates for the new car. As we can see in line 10, the code contains a loop over the identifiers of the cars.

5 Conclusions

In this paper we have described the creation of an animation layer for an elevator controller system case study. We consider that the animation layer is controlled by a CPN model, which has some additional elements that are specific to control the animation layer. The created animation is used during the validation task.

The idea of use an animation of a CPN model for the requirements validation is not new, for example Machado et al. present an approach to support the validation of workflow requirements for the interaction between people for a case study from a real project where animation were used [15]. Although these authors also consider that the CPN models are obtained from sequence diagrams, with this work we want to improve the mapping from sequence diagrams into the CPN model in order to support the parallel execution of many scenarios. We consider that the case study of Elevator controller is an example where the parallel execution of many scenarios is useful.

The CPN model is obtained from sequences diagrams that represents a set of scenario descriptions present in the use case behaviour. This CPN model explicitly model the entities on the environment that are important for the animation of the elevator controller system. It is possible to execute a given scenario in the CPN model for an initial state of the environment selected by the user. The mechanisms that were added to the CPN model to manipulate the animation are easily identified, thus it can be eliminated from the CPN model allowing the reuse of the CPN model in other tasks of the development process.

The animation layer consists on a representation of the problem domain in a user-friendly language, where the relevant entities of the system under development have a graphic representation with associated animations to represent the different behaviour of the entity. We use the SceneBeans tool to create the animation layer, which is defined using a XML-based file format. To assist on the management of XML-based code we present a script that permits an abstraction from the details associated to the usage of XML tags.

References

1. Dulac, N., Viguier, T., Leveson, N., Storey, M.A.: On the use of visualization in formal requirements specification. In: Proc. of IEEE Joint Int. Conf. on Requirements Engineering. (2002) 71–80
2. Wieringa, R.J.: Design Methods for Reactive Systems: Yourdon, Statemate, and the UML. Morgan Kaufmann (2003)
3. Jensen, K.: Coloured Petri Nets — Basic Concepts, Analysis Methods and Practical Use. Volume 1-3. Monographs in Theoretical Computer Science. EATCS Series. Springer (1992-97)
4. Jensen, K., Kristensen, L.M., Wells, L.: Coloured Petri Nets and CPN Tools for Modelling and Validation of Concurrent Systems. Int. Journal on Software Tools for Technology Transfer (STTT) **9**(3-4) (June 2007) 213–54
5. CPN Tools. Online: www.daimi.au.dk/CPNtools.
6. Blanco, R.M.: Requirements Specification for an Elevator Controller. Technical report, School of Computer Science, University of Waterloo, Canada (2005)
7. SceneBeans. Online: www-dse.doc.ic.ac.uk/Software/SceneBeans/.
8. Magee, J., Pryce, N., Giannakopoulou, D., Kramer, J.: Graphical animation of behavior models. In: Proc. of the 22nd Int. Conference on Software Engineering (ICSE'00), New York, NY, USA, ACM Press (2000) 499–508
9. Thomas, D., Fowler, C., Hunt, A.: Programming Ruby: The Pragmatic Programmers' Guide, Second Edition. Pragmatic Bookshelf (October 2004)
10. Ruby Programming Language. Online: www.ruby-lang.org/en/.
11. Ruby: REXML. Online: www.germane-software.com/software/rexml/.
12. Westergaard, M., Lassen, K.B.: The BRITNeY Suite Animation Tool. In Springer-Verlag, ed.: Proc. of 27th Int. Conference on Applications and Theory of Petri Nets (ICATPN'06). Volume 4024 of LNCS. (2006) 331–40
13. Westergaard, M.: The BRITNeY Suite: A Platform for Experiments. In: 7th Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools (CPN 2006). (2006)
14. Ribeiro, O.R., Fernandes, J.M.: Some Rules to Transform Sequence Diagrams into Coloured Petri Nets. In: 7th Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools (CPN 2006). (2006)
15. Machado, R.J., Lassen, K.B., Oliveira, S., Couto, M., Pinto, P.: Requirements Validation: Execution of UML Models with CPN Tools. Int. Journal on Software Tools for Technology Transfer (STTT) **9**(3-4) (June 2007) 353–369