A Tutorial on the Anemone Workbench: Solving the Rush Hour Problem

Jean-Marie Jacquet[a,*], Manel Barkallah[a]

[a] *Nadi Research Institute, Faculty of Computer Science, University of Namur*
Rue Grandgagnage 21, Namur, Belgium

Abstract

Although many research efforts have been spent on the theory and implementation of data-based coordination languages, not much effort has been devoted to constructing programming environments to analyze and reason over programs written in these languages. Recently the authors have proposed a workbench, named Anemone, for describing concurrent systems using a Linda-like language, for animating them and for reasoning on them using a reachability logic. This paper illustrates the use of the Anemone workbench by showing how it can be used to solve a traditional puzzle, the Rush Hour Problem.

Keywords: coordination languages, Bach, animation, verification

1. Introduction

In order to ease the development of interactive and distributed systems, Gelernter and Carriero have advocated in [1] to clearly separate the interactional and the computational aspects of software components. Their claim has been supported by the design of a model, Linda [2], originally presented as a set of inter-agent communication primitives which may be added to almost any programming language. Besides process creation, this set includes primitives for adding, deleting, and testing the presence/absence of data in a shared dataspace.

A number of other models, now referred to as coordination models, have been proposed afterwards (see eg [3, 4, 5, 6, 7, 8, 9, 10]). However, although many pieces of work have been devoted to the proposal of new languages, semantics and implementations, just a few articles (eg [11, 12, 13, 14]) have addressed the concerns of constructing programs in coordination languages in practice, in particular in checking that what is described by programs actually corresponds to what has to be modeled.

In [15], the authors have described a workbench, named Scan, to reason over Linda-like languages, with a particular focus on three goals:

*Corresponding author
Email addresses: jean-marie.jacquet@unamur.be (Jean-Marie Jacquet), manel.barkallah@unamur.be (Manel Barkallah)

Figure 1: Rush Hour Problem. On the left part, the game as illustrated at https://www.michaelfogleman.com/rush. On the right part, the game modeled as a grid of 6×6, with cars and trucks depicted as rectangles of different colors.

- to allow the user to understand the meaning of instructions written in Linda-like languages, by showing how they can be executed step by step and how the contents of the shared space, central to coordination languages, can be modified so as to release suspended processes;
- to allow the user to better grasp the modeling of real-life systems, by connecting agents to animations representing the evolution of the modeled system;
- to allow the user to check properties by model checking reachability formulae and by producing traces that can be replayed as evidences of the establishment of the formulae.

In this paper, we illustrate the use of an extension of the tool, named Anemone, by modeling a classical puzzle, the Rush Hour Problem. This game, illustrated in Figure 1, consists in moving cars and trucks on a 6×6 grid, according to their direction, such that the red car can exit. It can be formulated as a coordination problem by considering cars and trucks as autonomous agents which have to coordinate on the basis of free places.

The rest of this paper is organized as follows. Section 2 describes a model of the Rush Hour puzzle in the Multi-Bach process algebra on which Anemone is based. Section 3 sketches how this model can be processed by employing Anemone. Section 4 draws our conclusion and an appendix sums up the code obtained. It is worth noting that a companion video is available on the github repository of the Anemone project, accessible at `https://github.com/UNamurCSFaculty/anemone`.

2. A Model in Multi-Bach

Modeling the Rush Hour problem involves specifying data, processes describing the agents as well as a formula in temporal logic to check whether the problem is solvable and, in this case, to produce a trace illustrating a solution.

We turn subsequently to these tasks by using the Multi-Bach process algebra to be used with Anemone.

2.1. Definition of data

Following Linda, the Bach language [16, 17] uses four primitives for manipulating pieces of information : *tell* to put a piece of information on a shared space, *ask* to check its presence, *nask* to check its absence and *get* to check its presence and remove one occurrence. In its simplest version, named BachT, pieces of information consist of atomic tokens and the shared space, called the store, amounts to a multiset of tokens. Although in principle sufficient to code many applications, this is however too elementary in practice to code them easily. To that end, the Multi-Bach process algebra introduces more structured pieces of information which may employ sets defined by the user.

Sets. Sets are defined by associating an identifier with an enumeration of elements. In the case of the Rush Hour Problem, two sets are required: one to specify the columns and the rows (ranging from 1 to 6) and one to specify the colors of the vehicles. More specifically, our modeling declares the following sets:

Code: set definition

```
eset RCInt = { 1, 2, 3, 4, 5, 6 }.
     Colors = { yellow, green, blue, purple, red, orange }.
```

As indicated in Figure 1, we take the convention that the upper leftmost element of the grid is on the first row and on the first column.

It is worth noting that, for verification purposes, we deliberately restrict set definitions to finite sets. This limitation is of little concern in practice as most problems can be formulated by using finite sets. Moreover, we take profit of the enumeration of elements to induce an order between elements, thus implicitly defining the equality and inequalities relations. Strictly speaking our set declarations thus defines more than sets but also an order. Hence the `eset` tag to denote enumerated sets.

Syntax: set definition

Sets are defined by the keyword `eset` followed by a list of set declarations of the form

`set_identifier = set_enumeration.`

where `set_identifier` is a string of letters and digits, starting with an upper case letter, and where `set_enumeration` is a comma-separated list of strings composed of letters and digits, surrounded by the { and } brackets.

Si-terms. Structured pieces of information consist of flat tokens as well as expressions of the form $f(a_1, \cdots, a_n)$ where f is a functor and a_1, ..., a_n are set elements. Structured pieces of information are subsequently referred to as *si-terms* as well. In the Rush Hour example, it is sufficient to use the si-terms `free(i,j)` with i a row and j a column. Such si-terms indicate the places currently free on the grid, namely not occupied by a car or a truck.

Syntax: si-terms

Si-terms are either tokens, expressed as strings of letters and digits (possibly containing only letters or digits) or structured constructs of the form $f(t_1, \cdots, t_n)$ where $f, t_1, \ldots, t_n$ are such strings.

Maps. Anemone uses mapping equations as rewriting rules, from left to right in the aim of progressively reducing an expression into a set element. As an example, in the Rush Hour problem, assuming that trucks take three cells and are identified by the upper and left-most cell they occupy, the operation `down_truck` determines the cell to be taken by a truck moving down:

```
map down_truck : RCInt -> RCInt.
eqn down_truck(1) = 4. down_truck(2) = 5. down_truck(3) = 6.
```

Besides `down_truck`, our modeling of the Rush Hour problem requires to specify a similar mapping `down_car` for cars (taking two cells instead of three). It is also convenient to specify two additional mappings `right_truck` and `right_car` corresponding to horizontal movements. Finally, `pred` and `succ` mappings are defined to reflect the predecessor and successor functions. This lead us to the following code. Note that, as map signatures are not currently checked by Anemone, the map declarations are not made explicit.

Code: map rewriting

```
eqn down_car(1) = 3.      down_car(2) = 4.
    down_car(3) = 5.      down_car(4) = 6.
    down_truck(1) = 4.    down_truck(2) = 5.
    down_truck(3) = 6.
```

```
right_car(1) = 3.    right_car(2) = 4.
right_car(3) = 5.    right_car(4) = 6.
right_truck(1) = 4.  right_truck(2) = 5.
right_truck(3) = 6.

pred(2) = 1.  pred(3) = 2.  pred(4) = 3.
pred(5) = 4.  pred(6) = 5.
succ(1) = 2.  succ(2) = 3.  succ(3) = 4.
succ(4) = 5.  succ(5) = 6.
```

It is also worth noting from this example that mappings may be partially defined, with the responsibility put on the programmer to use them only when defined.

Syntax: map rewriting

Map rewritings are declared by using the `eqn` keyword together with equations of the form $s = t.$ where s and t are si-terms.

The mapping equations define a rewriting relation that rewrites si-terms to si-terms. It is subsequently denoted by $\rightsquigarrow$.

2.2. Agents

The primitives of Multi-Bach comprise the `tell`, `ask`, `nask` and `get` primitives already mentioned for Bach, which take si-terms as arguments. Primitives can be composed to form more complex agents by using traditional composition operators from concurrency theory: sequential composition, noted ';', parallel composition, noted '||', and non-deterministic choice, noted '+'. Multi-Bach adds two other mechanisms: conditional statements and generalized choice.

Conditional statements take the form $c \rightarrow a_1 \diamond a_2$, where c denotes a condition and a_1 and a_2 are agents. Conditions are obtained from elementary ones, thanks to the classical and, or and negation operators, denoted respectively by &, | and !. Elementary conditions are obtained by relating set elements or mappings on them by equalities (denoted =) or inequalities (denoted =, <, <=, >, >=).

Generalized choices are constructs of the form $\Sigma_{e \in S} A_e$ where A_e is an agent parameterized by variable e ranging in set S. Typically, this is obtained by introducing a si-term involving e. The execution is obtained as a generalization of a classical choice. All the instances of A_e obtained by letting e take all the values in S are offered as possible choices. An alternative which can perform a computation step is selected to perform the step of the generalized choice.

Procedures offer a very convenient manner to abstract from series of instructions. They are defined in Multi-Bach through the `proc` keyword by associating an agent with a procedure name. As in classical concurrency theory, it is assumed that the defining agents are guarded, in the sense that any call to a

procedure is preceded by the execution of a primitive or can be rewritten in such a form.

As an example, the behavior of a truck placed vertically at coordinates (r, c) and of color p can be described by a procedure `VerticalTruck(r,c,p)` stating that the truck either moves up or down. This leads to the following definition.

```
proc VerticalTruck(r: RCInt, c: RCInt; p: Colors) =
        VerticalTruckUp(r,c,p) + VerticalTruckDown(r,c,p).
```

For a truck to move up, it should be the case that it is not on the first row, namely that $r > 1$, and that the place above it (of coordinates $(pred(r), c)$) is free. In view of the definition of the free places in terms of the si-terms `free` placed on the store, this latter condition amounts to checking that the operation `get(free(pred(r),c))` succeeds. In that case, the truck moves up by liberating the last place of the grid it occupies, namely by telling `free(succ(succ(r)),c)`, and by calling recursively the `VerticalTruck` procedure with the new coordinates $(pred(r), c)$ and the same color p. The following code results from this reasoning:

```
proc VerticalTruckUp(r: RCInt, c: RCInt, p: Colors) =
        (r>1 & r<5) ->
          ( get(free(pred(r),c));
            tell(free(succ(succ(r)),c);
            VerticalTruck(pred(r),c,p) ).
```

It is worth observing that

```
(r>1 & r<5) -> ...
```

denotes a conditional statement while the ';' operator (denoting a sequential composition) forces to execute the primitive `get(...)` first, then the primitive `tell(...)` and finally the call `Verticaltruck(...)`. Note that, with respect to the above description, we have added the condition $r < 5$ to make sure that the construct $succ(succ(r))$ can be rewritten.

The procedure `VerticalTruckDown` can be coded similarly:

```
proc VerticalTruckDown(r: RCInt, c: RCInt) =
        (r>=1 & r<4) ->
          get(free(down_truck(r),c));
          tell(free(r,c));
          VerticalTruck(succ(r),c)).
```

The following code sums up the pieces of code for the vertical truck. To avoid introducing too many procedures, we have replaced the call to `VerticalTruckUp` and `VerticalTruckDown` by their definitions. Moreover to indicate the movement of the truck a special action `MoveTruck` has been introduced as a procedure call. The corresponding procedure will be defined later after animations have been introduced.

Code: the VerticalTruck procedure

```
proc VerticalTruck(r: RCInt, c: RCInt, p: Colors) =
        ( (r>1 & r<5) ->
            ( get(free(pred(r),c));
              MoveTruck(p,pred(r),c);
              tell(free(succ(succ(r)),c));
              VerticalTruck(pred(r),c,p)) )
        +
        ( (r>=1 & r<4) ->
            ( get(free(down_truck(r),c));
              MoveTruck(p,succ(r),c);
              tell(free(r,c));
              VerticalTruck(succ(r),c,p)) ).
```

Note that the execution of the choice (denoted by '+') is made on the first action of the alternatives. It is thus the possibility of performing the primitives `get(free(pred(r),c))` or `get(free(down_truck(r),c))` which determines the behavior of the trucks under consideration.

The code for vertical cars, horizontal trucks and horizontal cars is written similarly. It is provided in appendix. Using them, the yellow and green trucks of Figure 1 and the red and orange cars of Figure 1 can be defined as:

Code: some trucks and cars

```
proc YellowTruck = VerticalTruck(1,1,yellow).
     GreenTruck = HorizontalTruck(6,3,green).
     RedCar = HorizontalCar(3,2,red).
     OrangeCar = VerticalCar(5,6,orange).
```

As illustrated above, procedures may be defined recursively. This allows to code the initialization of free places through a double recursion: one on the rows and one on the column. The code is listed below. Given a row r and a (current) column c, the procedure `FreePlacesForRow(r,c)` tells the si-terms $free(r, y)$ for all columns y between c and 6. This is achieved by telling first `free(r,c)` and then performing a recursive call `FreePlacesForRow(r,succ(c))` in case $c < 6$. Based on this procedure, procedure `FreeRows(r)` proceeds similarly by calling `FreePlacesForRow(r,1)` to create the free places corresponding to row r and, if $r < 6$ by calling recursively `FreeRows(succ(r))` to treat the following rows. Note that the two procedure calls are put in parallel (through the || operator) as there is no reason to order them.

Code: free places

```
proc FreeRows(r: RCInt) =
        (r<6) -> ( FreePlacesForRow(r,1) ||
                        FreeRows(succ(r)) )
        +
        (r=6) -> ( FreePlacesForRow(r,1) ).
     FreePlacesForRow(r: RCInt,c: RCInt) =
        (c<6) -> ( tell(free(r,c));
                        FreePlacesForRow(r,succ(c)) )
        +
        (c=6) -> ( tell(free(r,c)) ).
```

Places occupied by the cars and trucks need to be removed. This may be achieved for the vertical trucks by the following procedure.

Code: places occupied by vertical trucks

```
proc VTruckPlaces(r: RCInt, c: RCInt) =
        (r<5) -> get(free(r,c));
                 get(free(succ(r),c));
                 get(free(succ(succ(r)),c)).
```

Procedures for horizontal truck, vertical and horizontal cars can be coded similarly. The code is in appendix. Following the above definition of yellow and green trucks as well as red and orange cars, we can define the placement of these trucks and cars as follows.

Code: places occupied by some trucks and cars

```
proc YellowTruckPlaces = VTruckPlaces(1,1).
     GreenTruckPlaces = HTruckPlaces(6,3).
     RedCarPlaces = HCarPlaces(3,2).
     OrangeCarPlaces = VCarPlaces(5,6).
```

As regards syntax, procedure definitions take the following form.

$$(\mathbf{T}) \quad \frac{t \rightsquigarrow u, u \text{ closed}}{\langle\, tell(t) \mid \sigma \,\rangle \longrightarrow \langle\, E \mid \sigma \cup \{u\} \,\rangle}$$

$$(\mathbf{A}) \quad \langle\, ask(t) \mid \sigma \cup \{u\} \,\rangle \longrightarrow \langle\, E \mid \sigma \cup \{u\} \,\rangle$$

$$(\mathbf{G}) \quad \langle\, get(t) \mid \sigma \cup \{u\} \,\rangle \longrightarrow \langle\, E \mid \sigma \,\rangle$$

$$(\mathbf{N}) \quad \frac{t \rightsquigarrow u, u \notin \sigma, u \text{ closed}}{\langle\, nask(t) \mid \sigma \,\rangle \longrightarrow \langle\, E \mid \sigma \,\rangle}$$

Figure 2: Transition rules for the primitives

Syntax: procedure definition and calls

Procedure definitions are declared by using the `proc` keyword together with equations of the form *proc_name* = *proc_decl*. where

- *proc_name* is of the form $p(a_1, \cdots, a_n)$ with
  - p a string of letters and digits, starting with an upper case letter
  - $a_1, \ldots, a_n$ formal arguments declared by constructs of the form $idLC : idUC$ associating a string $idLC$, composed of letters and digits starting with a lowercase letter, with a string $idUC$ of letters and digits starting with an uppercase letter, representing a set
- *proc_dec* is an agent defined from the primitives and the operators introduced in this subsection.

Procedure calls are constructs of the form $p(s_1, \cdots, s_n)$ with p a procedure name and $s_1, \ldots, s_n$ si-terms.

In summary, the statements of the Multi-Bach language, also called agents by abuse of language, consist of the statements A generated by the following grammar:

$$A \quad ::= \quad Prim \mid Proc \mid A \,;\, A \mid A \,||\, A \mid A \,+\, A \mid C \rightarrow A \diamond A \mid \Sigma_{e \in S} A_e$$

where $Prim$ represents a primitive, $Proc$ a procedure call, C a condition, e a variable and S a set.

The operational semantics of primitives and complex agents are respectively defined through the transition rules of Figures 2 and 3. Configurations consist of agents (summarizing the current state of the agents) and a multiset of si-terms (denoting the current state of the store). In order to express the termination of the computation of an agent, the set of agents is extended by a special terminating symbol E that can be seen as a completely computed agent. For

$$(\mathbf{S}) \quad \frac{\langle A \mid \sigma\rangle \longrightarrow \langle A' \mid \sigma'\rangle}{\langle A\ ;\ B \mid \sigma\rangle \longrightarrow \langle A'\ ;\ B \mid \sigma'\rangle}$$

$$(\mathbf{P}) \quad \frac{\langle A \mid \sigma\rangle \longrightarrow \langle A' \mid \sigma'\rangle}{\begin{array}{c}\langle A \mid\mid B \mid \sigma\rangle \longrightarrow \langle A' \mid\mid B \mid \sigma'\rangle \\ \langle B \mid\mid A \mid \sigma\rangle \longrightarrow \langle B \mid\mid A' \mid \sigma'\rangle\end{array}}$$

$$(\mathbf{C}) \quad \frac{\langle A \mid \sigma\rangle \longrightarrow \langle A' \mid \sigma'\rangle}{\begin{array}{c}\langle A\ +\ B \mid \sigma\rangle \longrightarrow \langle A' \mid \sigma'\rangle \\ \langle B\ +\ A \mid \sigma\rangle \longrightarrow \langle A' \mid \sigma'\rangle\end{array}}$$

$$(\mathbf{Gc}) \quad \frac{\langle A_f \mid \sigma\rangle \longrightarrow \langle A' \mid \sigma'\rangle,\ \text{for some } f \in S}{\langle \Sigma_{e\in S} A_e \mid \sigma\rangle \longrightarrow \langle A' \mid \sigma'\rangle}$$

$$(\mathbf{Co}) \quad \frac{\models C,\ \langle A \mid \sigma\rangle \longrightarrow \langle A' \mid \sigma'\rangle}{\begin{array}{c}\langle C \rightarrow A \diamond B \mid \sigma\rangle \longrightarrow \langle A' \mid \sigma'\rangle \\ \langle !C \rightarrow B \diamond A \mid \sigma\rangle \longrightarrow \langle A' \mid \sigma'\rangle\end{array}}$$

$$(\mathbf{Pc}) \quad \frac{P(\overline{x}) = A, \langle A[\overline{x}/\overline{u} \mid \sigma\rangle \longrightarrow \langle A' \mid \sigma'\rangle}{\langle P(\overline{u}) \mid \sigma\rangle \longrightarrow \langle A' \mid \sigma'\rangle}$$

Figure 3: Transition rules for the operators

uniformity purposes, we abuse the language by qualifying E as an agent. To meet the intuition, we always rewrite agents of the form $(E; A)$, $(E \mid\mid A)$ and $(A \mid\mid E)$ as A.

Rules of Figure 2 follow the intuitive description of the primitives. Note that before being processed, the si-term t is rewritten in u by means of the rewriting relation $\rightsquigarrow$. Rules of Figure 3 are quite classical. Rules (S), (P) and (C) provide the usual semantics for sequential, parallel and choice compositions. Rule (Gc) generalizes rule (C) for multiple alternatives obtained by variable e ranging over set S. As expected, rule (Co) specifies that the conditional instruction $C \rightarrow A \diamond B$ behaves as A if condition C can be evaluated to true and as B otherwise. Note that the notation $\models C$ is used to denote the fact that C evaluates to true. Rule (Pc) makes procedure call $P(\overline{u})$ behave as the agent A defining procedure P with the formal arguments $\overline{x}$ replaced by the actual ones $\overline{u}$.

2.3. Animations

Animations are obtained in a twofold manner: on the one hand, by describing the scene to be painted and, on the other hand, by primitives to place widgets, to move them, to make them appear or disappear, and more generally to change their attributes.

Several scenes may be defined in Anemone. The description of a scene consists in specifying the size of the canvas to be used by the animation, the background image of the animation, a series of images to be used, and a series of

widgets. A widget in turn involves the definition of attributes, the way they are displayed and various initial values for them. Besides user-defined attributes, six special attributes are predefined: the current coordinates `wdX` and `wdY` of the widget, the rotation angle `wdA` of the widget, the scale factor `wdF` of the widget, the visibility `wdV` of the widget, and the layer `wdL` of the widget.

Our modeling of the Rush Hour problem necessitates a single scene. It is subsequently named `rhScene`. It has been implemented by taking a picture and extracting trucks and cars from it. As a result, the size of the scene is 562 by 618. It uses only one layer named `top`. Several images are used. We report in the following code on four only. The first one defines the background. The second image illustrates the green truck. The two other images depict the red car according to the fact that it is in the grid or outside the grid. Note that file names are given with respect to the path in which Anemone is executed.

The scene is also composed of a series of widgets. The following code only reports on two. One is related to the green truck. It basically states that it has to be displayed by using the image of the green truck and is initially placed on the top layer. The second widget is similar for the red car, with the difference that it has to be displayed by the image corresponding to its position on the grid of the game. Note that we may have defined the initial place of the trucks and the cars in the `init` section of the widgets. However, for generality purposes, we have preferred to do so in the code written for the initialization. Note also that, by default, predefined attributes receive the following values: `wdX` = `wdY` = `wdA` = `wdV` = 0, `wdF` = 1 and `wdL` is the first element listed in `layers`.

Code: description of the scene

```
scene rhScene = {

    size = (562,618).
    layers = { top }.

    background = loadImage(Images/rh_empty.png).
    green_truck_img = loadImage(Images/green_truck.png).
    red_car_img_in = loadImage(Images/red_car_in.png).
    red_car_img_out = loadImage(Images/red_car_out.png).
    ...
```

Code: description of the scene (cont'd)

```
    ...
    widget green_truck = {
       display = { green_truck_img }
       init = { wdL = top. }
    }

    widget red_car = {
       attributes = { position in RCPlaces. }
       display = {
          position = in_grid -> red_car_img_in.
          position = out_grid -> red_car_img_out.
       }
       init = { wdL = top.
                position = in_grid. }
    }

    ...
}.
```

Syntax: scene definition

A scene is declared by using the `scene` keyword together with an equation of the form *scene_name* = *scene_description*.

The scene name consists of a string of letters and digits starting with a lower case letter. The scene description consists of a series of equalities defining the size of the scene, the layers, the background image, a series of images as well as a series of widgets, declared by means of the `widget` keyword. A widget in turn defines a set of attributes, by means of the `attributes` keyword, conditions on how to display the widget, thanks to the `display` keyword, and initial values thanks to the `init` keyword. Except for set identifiers, all the identifiers are strings of letters and digits starting with a lower case letter. Conditions guiding the display of widgets take the form of conditional statements defined in Subsection 2.2. The canvas size and coordinates are expressed in pixels, with $(0, 0)$ being the upper-left corner of the canvas.

Scenes and widgets are manipulated by means of two primitives:

- `draw_scene(s)`: to draw the scene identified by s

- `att(x,w,s,v)`: to give value v to the attribute x of the widget identified by w on scene s.

As some predefined attributes are so frequently used, a few primitives have been added to cope more declaratively with these attributes, although they are actually syntatic suggar for an initialization or the use of the `att` primitive. They consist of:

- `place_at(w,s,x,y)`: to place the widget identified by w on scene s at the coordinates (x, y)
- `move_to(w,s,x,y)`: to move the widget identified by w on scene s from its current position to the new coordinates (x, y)
- `hide(w,s)`: to hide the widget identified by w on scene s
- `show(w,s)`: to make appear the widget identified by w on scene s
- `layer(w,s,l)`: to place the widget identified by w on scene s on the layer l of that scene.

All these primitives are added to the various versions of the `tell`, `ask`, `get` and `nask` primitives of the previous subsections.

It is worth observing that the map constructs allow to declare coordinates in a symbolic manner. This makes it easy to specify the position of the widgets. In the Rush Hour example, by using (x, y) to denote respectively the column and the row occupied by the leftmost uppermost cell occupied by a vehicle, one may translate coordinates expressed logically in rows and columns of the grid in pixel positions as follows:

```
eqn x_vehicle(1) = 55.  x_vehicle(2) = 130. x_vehicle(3) = 210.
    x_vehicle(4) = 285. x_vehicle(5) = 360. x_vehicle(6) = 435.

    y_vehicle(1) = 80.  y_vehicle(2) = 155. y_vehicle(3) = 230.
    y_vehicle(4) = 310. y_vehicle(5) = 385. y_vehicle(6) = 460.
```

Using such declarations, one may then place the yellow truck by executing `place_at(yellow_truck,rhScene,x_vehicle(1),y_vehicle(1))`. More generally, placing a truck at a place can be coded as follows:

Code: placing trucks

```
proc PlaceTruck(r: RCInt, c: RCInt, p: Colors) =
        ( p = yellow -> place_at(yellow_truck,rhScene,
                                 x_vehicle(c),y_vehicle(r)) )
        +
        ( p = green -> place_at(green_truck,rhScene,
                                x_vehicle(c),y_vehicle(r)) )
        +
```

```
        ( p = blue -> place_at(blue_truck,rhScene,
                               x_vehicle(c),y_vehicle(r)) )
      +
        ( p = purple -> place_at(purple_truck,rhScene,
                               x_vehicle(c),y_vehicle(r)) ).
```

Placing cars proceeds similarly. Moreover, moving cars and trucks can be coded in a similar way, as illustrated below.

Code: moving trucks

```
proc MoveTruck(r:Rows, c: Cols, p: Colors) =
        ( p = yellow -> move_to(yellow_truck,rhScene,
                               x_vehicle(c),y_vehicle(r)) )
      +
        ( p = green -> move_to(green_truck,rhScene,
                               x_vehicle(c),y_vehicle(r)) )
      +
        ( p = blue -> move_to(blue_truck,rhScene,
                               x_vehicle(c),y_vehicle(r)) )
      +
        ( p = purple -> move_to(purple_truck,rhScene,
                               x_vehicle(c),y_vehicle(r)) ).
```

2.4. Processes as active data

An interesting feature of Linda concerns the perception of computation threads as active data. Such a data is put on the shared tuple space as an object to be evaluated. In our setting, procedure definitions provide us with a mechanism to reach such a goal. Indeed, telling a procedure call may be interpreted as creating a new thread of execution, parallel to the thread under consideration. As a procedure call is a string, we may use it to refer to the thread. Moreover, as a procedure call also resembles a si-term, we may consider it as a new form of data.

Consequently, telling several times a same procedure call creates several threads with the same name, in a similar way that the same si-term being told several times induces several occurrences on the store. Following the analogy, asking or getting a procedure call amounts to checking the presence of a thread corresponding to that call, and in the case of the get of removing one of the threads associated with the call. Moreover, performing a nask primitive on a procedure name amounts to checking the absence of a thread corresponding to the procedure call. Note that terminated processes are considered in a special stopped state but are not removed from the multiset of procedure calls.

As a result, the Multi-Bach process algebra incorporates four new primitives, `tellp(Proc)`, `askp(Proc)`, `naskp(Proc)` and `getp(Proc)`, to respectively create a thread in charge of computing the procedure call `Proc`, to test for its presence or absence, and to remove one thread computing `Proc`. In the Rush Hour problem, they can be used to launch the execution of a process associated with a truck or a car. For instance, `tellp(YellowTruck)` creates a new thread for the yellow truck, identified by the string 'YellowTruck'. Executing `getp(YellowTruck)` kills this thread, whatever point of execution the thread has reached.

2.5. Rules

Illustrating computations forces us not only to capture events directly produced by the computations but also to cope with events being caused indirectly by these computations. For instance, in the Rush Hour problem, if the red car indicates that it leaves the grid by placing the `out` token on the store, one may like to deduce from it that the puzzle is solved and that the attribute `position` of the red car widget has to move from `in_grid` to `out_grid`.

To that end, following [16, 18], Multi-Bach uses rules of the form

$$+t, -u \longrightarrow +v, -w$$

asserting that the presence of t and the absence of u implies the presence of v and the absence of w. More generally such rules may involve several positively and negatively marked objects both in the left and right parts of the rules. They are to be interpreted as the fact that any combination of objects from the left-hand side leads to the addition of positively marked objects in the right-hand side of the rule and the removal of one of the occurrences of objects negatively marked in this right-hand side. For instance, with respect to the above rule, a new occurrence of t or a removal of u leads to the addition of v and the removal of (one occurrence of) w.

Rules are applied as long as possible after each change on the blackboard. It is worth noting that, following the analogy between passive si-terms and active processes, objects of rules are to be understood as either si-terms or processes.

Concretely, Multi-Bach handles rules in a twofold manner. First, rules are declared by associating a name to an implication of the form mentioned above. Second, continuing along the analogy between passive and active data, rules are activated by telling them through the `tellr` primitive taking as argument the considered rule name. Rules are then checked for presence or absence by means of the primitives `askr` and `naskr` respectively. The primitive `getr(rn)` is finally used to remove one occurrence of the rule named `rn`. This is illustrated by the following rule declaration which asserts that any time a si-term `moveCar(r,c,p)` appears on the store, for some color `p`, some row number `r` and some column number `c`, the procedure call `MoveCar(r,c,p)` should be triggered. To avoid the rule to be applied several times, the `moveCar(p,r,c)` si-term is also removed.

```
rule move_car =
    for r in Rows, c in Cols, p in Colors:
        +moveCar(p,r,c)  -->  +MoveCar(p,r,c), -moveCar(p,r,c).
```

Besides its declaration, this rule is activated by executing `tellr(move_car)`, for instance in the beginning of the whole computation.

In our modeling of the Rush Hour problem, we use another rule, defined as follows

Code: rule end of puzzle

```
rule end_puzzle =
   +out -->+puzzle_solved, +RedCarOut, -out.
```

with procedure `RedCarOut` defined as

Code: procedure RedCarOut

```
proc RedCarOut = att(position,red_car,rhScene,out_grid).
```

Syntax: rules

A rule is declared by using the keyword `rule` together with an equation of the form *rule_name* = *rule_description*. The rule name is a string of letters and digits, starting with a lower case letter. The rule description is a construct of the form $quant : l\_m\_objects -- > l\_m\_objects$ where *quant* quantifies variables (represented as strings of letter and digits, starting with a lower case letter) over sets and where *l_m_objects* lists si-terms or procedure calls, positively or negatively marked.

2.6. A fragment of temporal logic

Linear temporal logic is a logic widely used to reason over dynamic systems. The Anemone workbench uses a fragment of PLTL [19] with, as main goal, to check the reachability of states.

As usual, the logic used by Anemone relies on propositional state formulae. In our coordination context, these formulae are to be verified on the current contents of the store. Consequently, given a structured piece of information t, we introduce $\#t$ to denote the number of occurrences of t on the store and define as basic propositional formulae, equalities or inequalities combining algebraic expressions involving integers and number of occurrences of structured pieces

of information. An example of such a basic formula is $\#free(1,1) = 1$ which states that the cell of coordinates $(1,1)$ is free.

Propositional state formulae are built from these basic formulae by using the classical propositional connectors. As particular cases, we use *true* and *false* to denote propositional formulae that are respectively always true and false. Such formulae are in fact shorthands to denote respectively $p \vee \neg p$ and $p \wedge \neg p$, for some basic propositional formula p.

The fragment of temporal logic used in Anemone is then defined by the following grammar :

$$TF ::= PF \mid Next\ TF \mid PF\ Until\ TF$$

where PF is a propositional formula. As an example, if the Rush Hour puzzle is considered as solved by having the store containing `puzzle_solved`, a solution to the rush problem is obtained by verifying the formula

$$true\ Until\ (\#out = 1)$$

Such formulae being very often used, they are abbreviated in Anemone as

$$Reach\ (\#out = 1)$$

3. Processing with Anemone

The Anemone workbench is available as a `jar` file from the github repository of the Anemone project at the address: `https://github.com/UNamurCSFaculty/anemone`. The source code is also available on that repository. It is written in Scala [20] by using the Processing library [21].

The easiest way of launching the workbench is probably to save the Anemone jar file in a directory, say `myDir`, and to type in a terminal

```
java -jar myDir/anemone.jar
```

As depicted in Figure 4, this has the effect of launching what we call the Interactive Blackboard. It displays the current contents of the shared space and allows to interact directly through the `tell`, `get`, `filter`, `unfilter` and `clear` buttons. Moreover, it offers to create five types of processes.

The first two processes, named respectively *Interactive Agent* and *Autonomous Agent*, allow the user to enter instructions in Multi-Bach and to execute them. As depicted in Figure 5, windows of the first kind, perform computations step-by-step by letting the user choose which primitives to execute. In contrast, windows of the second type execute computations in one run or in a step-by-step manner but in both cases with the Anemone workbench deciding the primitives to be executed. It is worth noting that the execution in the windows are made in a parallel fashion, hence the name *agent* to indicate entities capable of concurrent activities. In both cases, the `abbrev` option provides a convenience to abbreviate complex instructions by hiding the non-executable part.

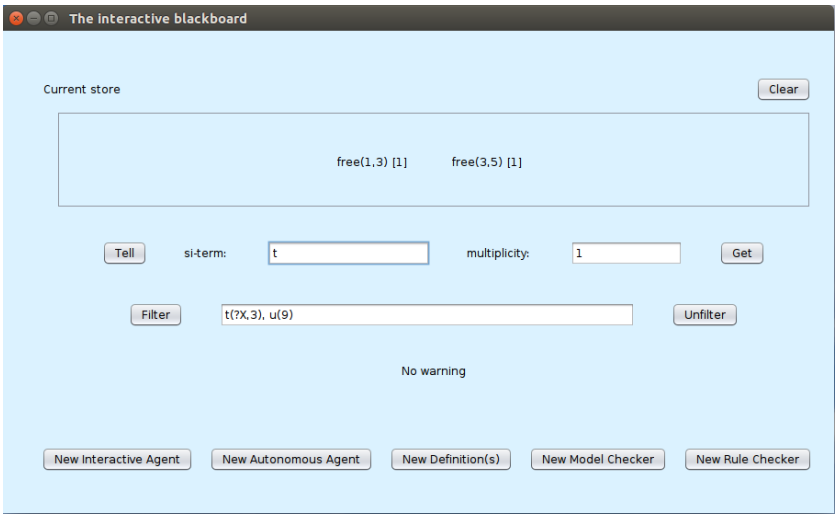

Figure 4: The interactive blackboard window

The facilities offered by the interactive and autonomous agents are well-suited to debug, at a low level, concurrent executions executed directly over the shared space, possibly deadlocking while waiting for unavailable data. However, they do not provide much abstraction to describe and reason on models. To that end, Anemone provides three other kinds of windows:

- a window to declare sets, procedures, scenes, and rules. It is launched by means of the `new definition(s)` button
- a window to reason on the execution of rules. It is launched by the `New rule checker` button.
- a window to model check properties. It is launched by the `New model checker` button.

The use of all these windows is illustrated by videos available on the github repository of the Anemone project. We refer the reader to them for further details.

4. Conclusion

This paper has illustrated the Anemone workbench by showing how it can be used to solve a traditional puzzle, the Rush Hour Problem. To that end, the puzzle has first been modeled in the Multi-Bach process algebra. It has then been suggested how it can be employed at three levels: (i) by executing in a step by step or automatic manner instructions while showing their impact on

(a) The interactive agent window

(b) The autonomous agent window

Figure 5: Interacting with the blackboard

the shared space, (ii) by illustrating computations through animations and (iii) by model checking properties by means of temporal formulae. A companion video further details how to proceed in practice.

References

[1] D. Gelernter, N. Carriero, Coordination Languages and Their Significance, Communications of the ACM 35 (2) (1992) 97–107.

[2] N. Carriero, D. Gelernter, Linda in Context, Communications of the ACM 32 (4) (1989) 444–458.

[3] M. Banville, SONIA: An Adaptation of LINDA for Coordination of Activities in Organisations, in: P. Ciancarini, C. Hankin (Eds.), Proceedings of the International Conference on Coordination Languages and Models, Vol. 1061 of Lecture Notes in Computer Science, Springer, 1996, pp. 57–74.

[4] K. D. Bosschere, J.-M. Jacquet, μ^2 Log: Towards Remote Coordination, in: P. Ciancarini, C. Hankin (Eds.), Proceedings of the International Conference on Coordination Languages and Models, Vol. 1061 of Lecture Notes in Computer Science, Springer, 1996, pp. 142–159.

[5] M. Cremonini, A. Omicini, F. Zambonelli, Coordination in Context: Authentication, Authorisation and Topology in Mobile Agent Applications, in: P. Ciancarini, A. Wolf (Eds.), Proceedings of the International Conference on Coordination Languages and Models, Vol. 1594 of Lecture Notes in Computer Science, Springer, 1999, p. 416.

[6] C.-L. Fok, G.-C. Roman, G. Hackmann, A Lightweight Coordination Middleware for Mobile Computing, in: R. De Nicola, G. Ferrari, G. Meredith (Eds.), Proceedings of the International Conference on Coordination Models and Languages, Vol. 2949, Springer, 2004, pp. 135–151.

[7] R. Tolksdorf, Coordinating Services in Open Distributed Systems with LAURA, in: P. Ciancarini, C. Hankin (Eds.), Proceedings of the International Conference on Coordination Languages and Models, Vol. 1061 of Lecture Notes in Computer Science, Springer, 1996, pp. 386–402.

[8] L. Bettini, V. Bono, R. De Nicola, G. Ferrari, D. Gorla, M. Loreti, E. Moggi, R. Pugliese, E. Tuosto, B. Venneri, The Klaim Project: Theory and Practice, in: C. Priami (Ed.), Proceedings of the International Workshop on Global Computing, Programming Environments, Languages, Security, and Analysis of Systems, Vol. 2874 of Lecture Notes in Computer Science, Springer, 2003, pp. 88–150.

[9] A. Murphy, G. Picco, G.-C. Roman, LIME: A Coordination Model and Middleware Supporting Mobility of Hosts and Agents, ACM Trans. Softw. Eng. Methodol. 15 (3) (2006) 279–328.

[10] R. De Nicola, T. Duong, O. Inverso, C. Trubiani, AErlang: Empowering Erlang with Attribute-based Communication, Science of Computer Programming 168 (2018) 71–93.

[11] N. Kokash, F. Arbab, Formal Design and Verification of Long-Running Transactions with Extensible Coordination Tools, IEEE Transactions on Services Computing 6 (2) (2013) 186–200.

[12] R. Cruz, J. Proença, ReoLive: Analysing Connectors in Your Browser, in: M. Mazzara, I. Ober, G. Salaün (Eds.), Proceedings of the STAF collocated workshops, Vol. 11176 of Lecture Notes in Computer Science, Springer, 2018, pp. 336–350.

[13] F. Calzolai, R. De Nicola, M. Loreti, F. Tiezzi, TAPAs: A Tool for the Analysis of Process Algebras, in: K. Jensen, W. van der Aalst, J. Billington (Eds.), Transactions on Petri Nets and Other Models of Concurrency, Vol. 5100 of Lecture Notes in Computer Science, Springer, 2008, pp. 54–70.

[14] L. Bettini, E. Merelli, F. Tiezzi, X-Klaim Is Back, in: M. Boreale, F. Corradini, M. Loreti, R. Pugliese (Eds.), Models, Languages, and Tools for Concurrent and Distributed Programming: Essays Dedicated to Rocco De Nicola on the Occasion of His 65th Birthday, Vol. 11665 of Lecture Notes in Computer Science, Springer, 2019, pp. 115–135.

[15] J.-M. Jacquet, M. Barkallah, Scan: A Simple Coordination Workbench, in: H. Riis Nielson, E. Tuosto (Eds.), Proceedings of the 21st International Conference on Coordination Models and Languages, Vol. 11533 of Lecture Notes in Computer Science, Springer, 2019, pp. 75–91.

[16] D. Darquennes, J.-M. Jacquet, I. Linden, On Multiplicities in Tuple-Based Coordination Languages: The Bach Family of Languages and Its Expressiveness Study, in: G. D. M. Serugendo, M. Loreti (Eds.), Proceedings of the 20th International Conference on Coordination Models and Languages, Vol. 10852 of Lecture Notes in Computer Science, Springer, 2018, pp. 81–109.

[17] J.-M. Jacquet, I. Linden, Coordinating Context-aware Applications in Mobile Ad-hoc Networks, in: T. Braun, D. Konstantas, S. Mascolo, M. Wulff (Eds.), Proceedings of the first ERCIM workshop on eMobility, The University of Bern, 2007, pp. 107–118.

[18] J.-M. Jacquet, I. Linden, M. Staicu, Blackboard Rules: from a Declarative Reading to its Application for Coordinating Context-aware Applications in Mobile Ad Hoc Networks, Science of Computer Programming 115-116 (2016) 79–99.

[19] E. A. Emerson, Temporal and Modal Logic, in: Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics (B), Elsevier, 1990, pp. 995–1072.

[20] M. Odersky, L. Spoon, B. Venners, Programming in Scala, A comprehensive step-by-step guide, Artemis, 2016.

[21] C. Reas, B. Fry, Processing: A Programming Handbook for Visual Designers, The MIT Press, 2014.

Appendix A. Final code for the Rush Hour Problem

The appendix lists the code obtained for the Rush Hour Problem following sections 2 and 3. To the developments of these sections, we have added at the end of the code, two descriptions of two games.

Appendix A.1. Data declaration

```
eset RCInt = { 1, 2, 3, 4, 5, 6 }.
     Colors = { yellow, green, blue, purple, red, orange }.
     RCPlaces = { in_grid, out_grid }.

eqn down_car(1) = 3. down_car(2) = 4.
    down_car(3) = 5. down_car(4) = 6.
    down_truck(1) = 4. down_truck(2) = 5. down_truck(3) = 6.

    right_car(1) = 3. right_car(2) = 4.
    right_car(3) = 5. right_car(4) = 6.
    right_truck(1) = 4. right_truck(2) = 5. right_truck(3) = 6.

    pred(2) = 1. pred(3) = 2. pred(4) = 3.
    pred(5) = 4. pred(6) = 5.
    succ(1) = 2. succ(2) = 3. succ(3) = 4.
    succ(4) = 5. succ(5) = 6.

    img_truck(yellow) = yellow_truck.
    img_truck(green) = green_truck.
    img_truck(blue) = blue_truck.
    img_truck(purple) = purple_truck.

    img_car(red) = red_car. img_car(green) = green_car.
    img_car(orange) = orange_car.

    x_vehicle(1) = 55.  x_vehicle(2) = 130. x_vehicle(3) = 210.
    x_vehicle(4) = 285. x_vehicle(5) = 360. x_vehicle(6) = 435.

    y_vehicle(1) = 80.  y_vehicle(2) = 155. y_vehicle(3) = 230.
    y_vehicle(4) = 310. y_vehicle(5) = 385. y_vehicle(6) = 460.
```

Appendix A.2. Scene description

```
scene rhScene = {

    size = (562,618).
    layers = { top }.

    background = loadImage(Images/rh_empty.png).
    yellow_truck_img = loadImage(Images/yellow_truck.png).
    green_truck_img = loadImage(Images/green_truck.png).
    blue_truck_img = loadImage(Images/blue_truck.png).
    purple_truck_img = loadImage(Images/purple_truck.png).
    red_car_img_in = loadImage(Images/red_car_in.png).
    red_car_img_out = loadImage(Images/red_car_out.png).
    green_car_img = loadImage(Images/green_car.png).
    orange_car_img = loadImage(Images/orange_car.png).

    widget yellow_truck = {
       display = { yellow_truck_img }
       init = { wdL = top. }
    }

    widget green_truck = {
       display = { green_truck_img }
       init = { wdL = top. }
    }

    widget blue_truck = {
       display = { blue_truck_img }
       init = { wdL = top. }
    }

    widget purple_truck = {
       display = { purple_truck_img }
       init = { wdL = top. }
    }

    widget red_car = {
       attributes = { position in RCPlaces. }
       display = {
          position = in_grid -> red_car_img_in.
          position = out_grid -> red_car_img_out.
       }
       init = {
          wdL = top.
          position = in_grid.
       }
    }
```

```
    widget green_car = {
       display = { green_car_img }
       init = { wdL = top. }
    }

    widget orange_car = {
       display = { orange_car_img }
       init = { wdL = top. }
    }

}.
```

Appendix A.3. Trucks and cars

```
proc VerticalCar(r: RCInt, c: RCInt, p: Colors) =
        ( (r>1 & r<6) -> ( get(free(pred(r),c));
                           MoveCar(pred(r),c,p);
                           tell(free(succ(r),c));
                           VerticalCar(pred(r),c,p) ))
        +
        ( (r>=1 & r<5) -> ( get(free(down_car(r),c));
                            MoveCar(succ(r),c,p);
                            tell(free(r,c));
                            VerticalCar(succ(r),c,p) )).

     VerticalTruck(r: RCInt, c: RCInt, p: Colors) =
        ( (r>1 & r<5) ->  ( get(free(pred(r),c));
                            MoveTruck(pred(r),c,p);
                            tell(free(succ(succ(r)),c));
                            VerticalTruck(pred(r),c,p) ))
        +
        ( (r>=1 & r<4) -> ( get(free(down_truck(r),c));
                            MoveTruck(succ(r),c,p);
                            tell(free(r,c));
                            VerticalTruck(succ(r),c,p) )).

     HorizontalCar(r: RCInt, c: RCInt, p: Colors) =
        ( (c>1 & c<6) ->  ( get(free(r,pred(c)));
                            MoveCar(r,pred(c),p);
                            tell(free(r,succ(c)));
                            HorizontalCar(r,pred(c),p) ))
        +
        ( (c>=1 & c<5) -> ( get(free(r,right_car(c)));
                            MoveCar(r,succ(c),p);
                            tell(free(r,c));
                            HorizontalCar(r,succ(c),p) ))
        +
        ( (r=3 & c=5 & p=red) -> ( tell(out); MoveCar(p,r,succ(c)) )).
```

```
HorizontalTruck(r: RCInt, c: RCInt, p: Colors) =
    ( (c>1 & c<5) ->  ( get(free(r,pred(c)));
                        MoveTruck(r,pred(c),p);
                        tell(free(r,succ(succ(c))));
                        HorizontalTruck(r,pred(c),p) ))
    +
    ( (c>=1 & c<4) -> ( get(free(r,right_truck(c)));
                        MoveTruck(r,succ(c),p);
                        tell(free(r,c));
                        HorizontalTruck(r,succ(c),p) )).
```

Appendix A.4. Auxiliary operations

```
proc MoveCar(r:RCInt, c: RCInt, p: Colors) =
        ( p = red -> move_to(red_car,rhScene,x_vehicle(c),y_vehicle(r)) )
        +
        ( p = green -> move_to(green_car,rhScene,x_vehicle(c),y_vehicle(r)) )
        +
        ( p = organge -> move_to(orange_car,rhScene,x_vehicle(c),y_vehicle(r)) ).

    MoveTruck(r:RCInt, c: RCInt, p: Colors) =
        ( p = yellow -> move_to(yellow_truck,rhScene,x_vehicle(c),y_vehicle(r)) )
        +
        ( p = green -> move_to(green_truck,rhScene,x_vehicle(c),y_vehicle(r)) )
        +
        ( p = blue -> move_to(blue_truck,rhScene,x_vehicle(c),y_vehicle(r)) )
        +
        ( p = purple -> move_to(purple_truck,rhScene,x_vehicle(c),y_vehicle(r)) ).

    PlaceTruck(r: RCInt, c: RCInt, p: Colors) =
        ( p = yellow -> place_at(yellow_truck,rhScene,x_vehicle(c),y_vehicle(r)) )
        +
        ( p = green -> place_at(green_truck,rhScene,x_vehicle(c),y_vehicle(r)) )
        +
        ( p = blue -> place_at(blue_truck,rhScene,x_vehicle(c),y_vehicle(r)) )
        +
        ( p = purple -> place_at(purple_truck,rhScene,x_vehicle(c),y_vehicle(r)) ).

    PlaceCar(r: RCInt, c: RCInt, p: Colors) =
        ( p = red -> place_at(red_car,rhScene,x_vehicle(c),y_vehicle(r)) )
        +
        ( p = green -> place_at(green_car,rhScene,x_vehicle(c),y_vehicle(r)) )
        +
        ( p = orange -> place_at(orange_car,rhScene,x_vehicle(c),y_vehicle(r)) ).

    ShowTruck(p: Colors) =
        ( p = yellow -> show(yellow_truck,rhScene) )
        +
        ( p = green -> show(green_truck,rhScene) )
```

```
    +
    ( p = blue -> show(blue_truck,rhScene) )
    +
    ( p = purple -> show(purple_truck,rhScene) ).

ShowCar(p: Colors) =
    ( p = red -> show(red_car,rhScene) )
    +
    ( p = green -> show(green_car,rhScene) )
    +
    ( p = orange -> show(orange_car,rhScene) ).
```

Appendix A.5. Place handling

```
InitFreePlaces = FreeRows(1).

FreeRows(r: RCInt) =
    ( (r<6) -> ( FreePlacesForRow(r,1) || FreeRows(succ(r)) ) )
    +
    ( (r=6) -> ( FreePlacesForRow(r,1) ) ).

FreePlacesForRow(r: RCInt,c: RCInt) =
    ( (c<6) -> ( tell(free(r,c)); FreePlacesForRow(r,succ(c)) ) )
    +
    ( (c=6) -> ( tell(free(r,c)) ) ).

VTruckPlaces(r: RCInt, c: RCInt) =
    (r<5) -> get(free(r,c));
             get(free(succ(r),c));
             get(free(succ(succ(r)),c)).

VCarPlaces(r: RCInt, c: RCInt) =
    (r<6) -> get(free(r,c));
             get(free(succ(r),c)).

HTruckPlaces(r: RCInt, c: RCInt) =
    (c<5) -> get(free(r,c));
             get(free(r,succ(c)));
             get(free(r,succ(succ(c)))).

HCarPlaces(r: RCInt, c: RCInt) =
    (c<6) -> get(free(r,c));
             get(free(r,succ(c))).
```

Appendix A.6. Car and truck instances

```
proc YellowTruck(r: RCInt, c: RCInt) = VerticalTruck(r,c,yellow).
     GreenTruck(r: RCInt, c: RCInt) = HorizontalTruck(r,c,green).
     BlueTruck(r: RCInt, c: RCInt) = HorizontalTruck(r,c,blue).
```

```
    PurpleTruck(r: RCInt, c: RCInt) = VerticalTruck(r,c,purple).

    RedCar(r: RCInt, c: RCInt) = HorizontalCar(r,c,red).
    GreenCar(r: RCInt, c: RCInt) = VerticalCar(r,c,green).
    OrangeCar(r: RCInt, c: RCInt) = VerticalCar(r,c,orange).

    YellowTruckAt(r: RCInt, c: RCInt) =
       VTruckPlaces(r,c); PlaceTruck(r,c,yellow); ShowTruck(yellow).
    GreenTruckAt(r: RCInt, c: RCInt) =
       HTruckPlaces(r,c); PlaceTruck(r,c,green); ShowTruck(green).
    BlueTruckAt(r: RCInt, c: RCInt) =
       HTruckPlaces(r,c); PlaceTruck(r,c,blue); ShowTruck(blue).
    PurpleTruckAt(r: RCInt, c: RCInt) =
       VTruckPlaces(r,c); PlaceTruck(r,c,purple); ShowTruck(purple).

    RedCarAt(r: RCInt, c: RCInt) =
       HCarPlaces(r,c); PlaceCar(r,c,red); ShowCar(red).
    GreenCarAt(r: RCInt, c: RCInt) =
       VCarPlaces(r,c); PlaceCar(r,c,green); ShowCar(green).
    OrangeCarAt(r: RCInt, c: RCInt) =
       VCarPlaces(r,c); PlaceCar(r,c,orange); ShowCar(orange).
```

Appendix A.7. High-level descriptions

```
proc InitScene = draw_scene(rhScene).

    Init_game_1 =
       InitScene; InitFreePlaces;
       YellowTruckAt(1,1); GreenTruckAt(6,3);
       BlueTruckAt(4,4); PurpleTruckAt(1,4);
       RedCarAt(3,2); GreenCarAt(4,3); OrangeCarAt(5,6);
       ( ( (YellowTruck(1,1) || GreenTruck(6,3)) ||
           (BlueTruck(4,4) || PurpleTruck(1,4)) ) ||
         ( RedCar(3,2) || (GreenCar(4,3) || OrangeCar(5,6)) ) ).

    Init_game_2 =
       InitScene; InitFreePlaces;
       PurpleTruckAt(2,6); RedCarAt(3,4);
       ( PurpleTruck(2,6) || RedCar(3,4) ).
```

Appendix A.8. Ending the game

```
proc RedCarOut = att(position,red_car,rhScene,out_grid).

rule end_puzzle =
        +out -->+puzzle_solved, +RedCarOut, -out.
```