Easy Graphical Animation and Formula Visualisation for Teaching B [1]

Michael Leuschel[2] Mireille Samia[3] Jens Bendisposto[4] Li Luo

Softwaretechnik und Programmiersprachen
Institut für Informatik, Heinrich-Heine-Universität Düsseldorf
Universitätsstr. 1, 40225 Düsseldorf, Germany

Abstract

ProB is being used for teaching the B-method. In this paper, we present two new features of ProB that we have introduced while teaching B. One feature allows a student (or an expert user) to graphically visualise any predicate as a tree. The underlying algorithm can deal with undefined subformulas and tries to provide useful feedback even for existentially quantified formulas which are false. This feature is especially useful to inspect unexpected invariant violations or operations which are unexpectedly enabled or disabled. The other feature enables a student or lecturer to easily and quickly write custom graphical state representations, to provide a better understanding of the model. With this method, one simply has to assemble a series of pictures and to write an animation function in B itself, which stipulates which pictures should be shown where depending on the current state of the model. As an additional side-benefit, writing the animation function in B itself is a good exercise for students.

Keywords: B-Method, Tool Support, Animation

1 Introduction

There are two main proof activities in B: *consistency checking*, which is used to show that the operations of a machine preserve the invariant, and *refinement checking*, which is used to show that one machine is a valid refinement of another. These activities are supported by tools, such as Atelier-B, B4Free, and the B-toolkit. In addition to the proof activities, it is increasingly being realised that validation of the initial specification is important to avoid deriving a "correct" implementation of an incorrect specification. This validation can come in the form of *animation*, e.g., to check that certain functionality is present in the specification. Another useful tool is *model checking* [6], whereby the specification can be systematically checked for properties expressed in a temporal logic. In previous work [9], the ProB animator

[1] This research is being carried out as part of the EU funded FP7 research project 214158: DEPLOY (Industrial deployment of advanced system engineering methods for high productivity and dependability).

[2] Email: leuschel@cs.uni-duesseldorf.de

[3] Email: samia@cs.uni-duesseldorf.de

[4] Email: bendisposto@cs.uni-duesseldorf.de

and model checker has been presented to support those activities. The tool can also be used to complement proof activities, as it supports automated consistency and refinement checking of B machines.

ProB [9] is used at several universities for teaching the B-method.[5] Usually the feedback of students is very positive, and we feel that animation helps the students get a better understanding of the B formalism. The students have also provided useful feedback about the tool, and over the years we have added many new features to make the tool more useful. One such feature, which we present in Section 2, allows the student (but also the industrial user) to inspect formulas and graphically visualise them as a tree. It is especially useful to analyse unexpected invariant violations or operations which are unexpectedly enabled or disabled.

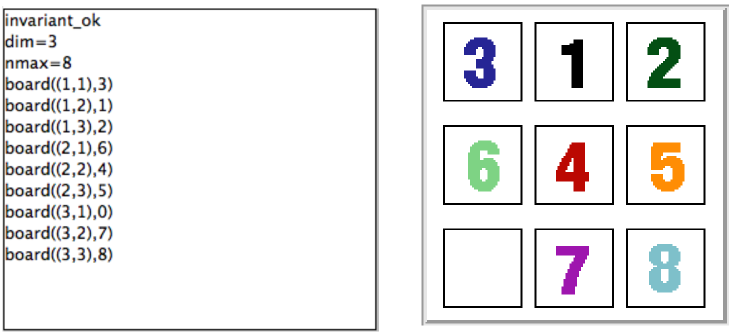
The possibility to see and to explore the behaviour of a formal model is often invaluable for students. ProB provides feedback about the current state, the enabled operations and can be used to visualise the statespace of a model [10]. However, sometimes a more graphical, domain-specific visualisation of the current state of a formal model is desirable, to give the student a better understanding of the model. In earlier work [3], we have presented a flash-based animation engine. However, this engine still requires careful setup and the development of gluing code together with Flash animations. It is thus usually too complicated for students to set up and use themselves, and even for lecturers the overhead can be prohibitive. In this paper, we provide a very simple but effective way of producing custom animations of a model. With this method, the student or lecturer simply has to assemble a series of pictures and has to write an animation function in B itself, which stipulates which pictures should be shown where depending on the current state of the system. As an additional side-benefit, writing the animation function in B itself is a good exercise for students.

Take for example the following B machine describing the well-known sliding 8-puzzle, where numbered tiles can be moved horizontally and vertically (into an empty square). The goal is to reach a configuration where all tiles are in order. On the left of Figure 1 you can see the non-graphical visualisation provided by ProB. It is not unreadable, but the graphical visualisation on the right is clearly much more inspiring and understandable. We now show how this animation can be achieved using our new version of ProB with very little effort.

```
MACHINE Puzzle8
DEFINITIONS INV == (board: ((1..dim)*(1..dim)) -->> 0..nmax);
GOAL == !(i,j).(i:1..dim & j:1..dim =>
           board(i|->j) = j-1+(i-1)*dim);
CONSTANTS dim, nmax
PROPERTIES dim:NATURAL1 &  dim=3 & nmax:NATURAL1 & nmax = dim*dim-1
VARIABLES board
INVARIANT INV
INITIALISATION board : (INV & GOAL)
OPERATIONS
   MoveDown(i,j,x) = PRE i:2..dim & j:1..dim &
      board(i|->j) = 0 & x:1..nmax & board(i-1|->j) = x
     THEN board := board <+ {(i|->j)|->x, (i-1|->j)|->0}
   END;
   MoveUp(i,j,x) = PRE i:1..dim-1 & j:1..dim &
      board(i|->j) = 0 & x:1..nmax & board(i+1|->j) = x
     THEN board := board <+ {(i|->j)|->x, (i+1|->j)|->0}
   END;
   MoveRight(i,j,x) = PRE i:1..dim & j:2..dim &
```

[5] For example, Besançon,Nantes in France; Southampton and Surrey in England; McMaster University, in Canada, Uppsala University in Sweden, and of course Düsseldorf in Germany.

```
      board(i|->j) = 0 & x:1..nmax & board(i|->j-1) = x
     THEN board := board <+ {(i|->j)|->x, (i|->j-1)|->0}
   END;
   MoveLeft(i,j,x) = PRE i:1..dim & j:1..dim-1 &
      board(i|->j) = 0 & x:1..nmax & board(i|->j+1) = x
     THEN board := board <+ {(i|->j)|->x, (i|->j+1)|->0}
   END
END
```



```
invariant_ok
dim=3
nmax=8
board((1,1),3)
board((1,2),1)
board((1,3),2)
board((2,1),6)
board((2,2),4)
board((2,3),5)
board((3,1),0)
board((3,2),7)
board((3,3),8)
```

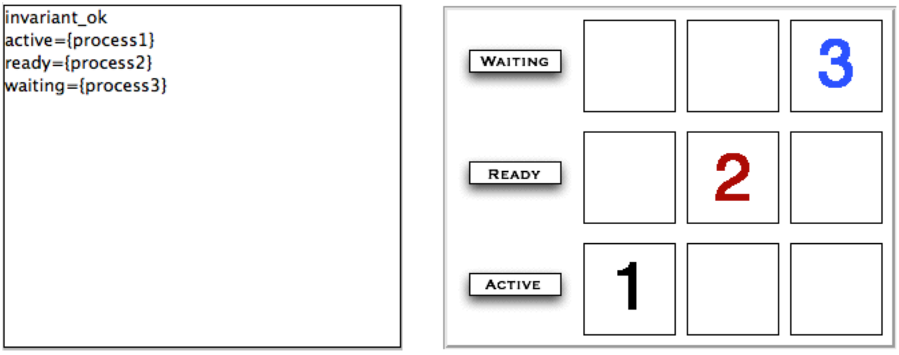
Fig. 1. Puzzle8 Non-Graphical and Graphical Visualisation

Another example is the scheduler from [8], whose code is repeated in Figure 2. Figure 3 shows both the non-graphical animation on the left, as well as the graphical animation obtained using our new tool, which clearly shows to the student and the user how the processes progress through the various stages.

```
MACHINE scheduler
SETS PID = {process1,process2,process3}
VARIABLES active, ready, waiting
INVARIANT
   active <: PID & ready <: PID &  waiting <: PID &
   (ready /\ waiting) = {} &
   active /\ (ready \/ waiting) = {} &
   card(active) <= 1 &
   ((active = {}) => (ready = {}))
INITIALISATION active := {} || ready := {} || waiting := {}
OPERATIONS
   new(pp) =
     SELECT pp : PID & pp /: active & pp /: (ready \/ waiting)
     THEN waiting := (waiting \/ { pp })
   END;
   del(pp) =
     SELECT pp : waiting
     THEN waiting := waiting - { pp }
   END;
   ready(rr) = SELECT rr : waiting
     THEN waiting := (waiting - {rr}) ||
      IF (active = {})
      THEN active := {rr}
      ELSE ready := ready \/ {rr}
      END
     END;
   swap = SELECT active /= {}
     THEN
      waiting := (waiting \/ active) ||
      IF (ready = {}) THEN active := {}
      ELSE ANY pp WHERE pp : ready
           THEN active := {pp} || ready := ready - {pp}
           END
      END
     END
END
```

Fig. 2. Scheduler Specification

We present this new feature for ProB in Section 3 along with typical examples from teaching, and apply it to a larger case study the size of a typical student project in Section 5.



```
invariant_ok
active={process1}
ready={process2}
waiting={process3}
```

Fig. 3. Scheduler Non-Graphical and Graphical Visualisation

2 Visualising Formulas

Animation has the purpose of identifying unexpected behaviours of a model. If an unexpected behaviour does occur, the user often would like to know more about the source of the problem.

- If the invariant is violated, one would like to know exactly which part of the invariant is violated and why.
- If an operation is unexpectedly not enabled, or unexpectedly enabled, one would like to know the reason.
- If the animator cannot find values for the constants which satisfy the properties of a machine, one would like to be able to locate the problematic properties.

For this, ProB had for quite some time the ability to inspect the invariant and also to debug the properties. However, this view was often not precise enough, and we have now implemented an algorithm which can take any B predicate or expression and translates it into a graphical representation that can be inspected by the user. We believe this feature to be especially useful for students and newcomers to B, but we also believe it to be important when animating complex specifications.

The algorithm basically uses the ProB interpreter to compute the value of an expression or the truth-value of a predicate. It then tries to decompose the expression or predicate into sub-expressions or -predicates. These are in turn recursively evaluated, until we reach sub-expressions or -predicates which can no longer be decomposed. The whole is then assembled into a graphical tree representation and rendered using the GraphViz package [2]. For example, Figure 4 contains a visualisation of the invariant of the scheduler machine, for the state already depicted earlier in Figure 3. For each expression, we have two lines of text: the first indicates the type of the node, i.e., the top-level operator. The second line gives the value of evaluating the expression. For predicates, the situation is similar, except that there is a third line with the formula itself and that the nodes are coloured: true predicates are green and false predicates are red.

Note that our algorithm also deals with undefined predicates. Those are rendered in orange. For example, the formula $x \in dom(f) \wedge f(x) = 1$, where f is the empty function (and x some value aa) would be visualised as in Figure 5.

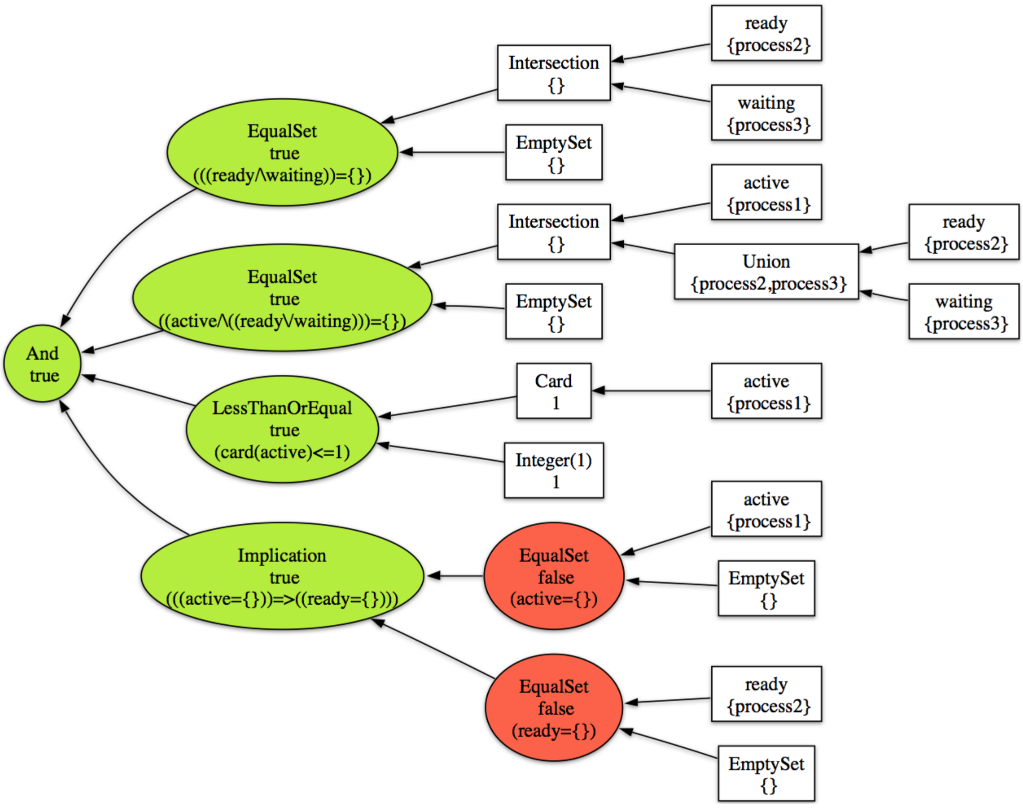


Fig. 4. Visualising the invariant of the scheduler for state of Figure 3

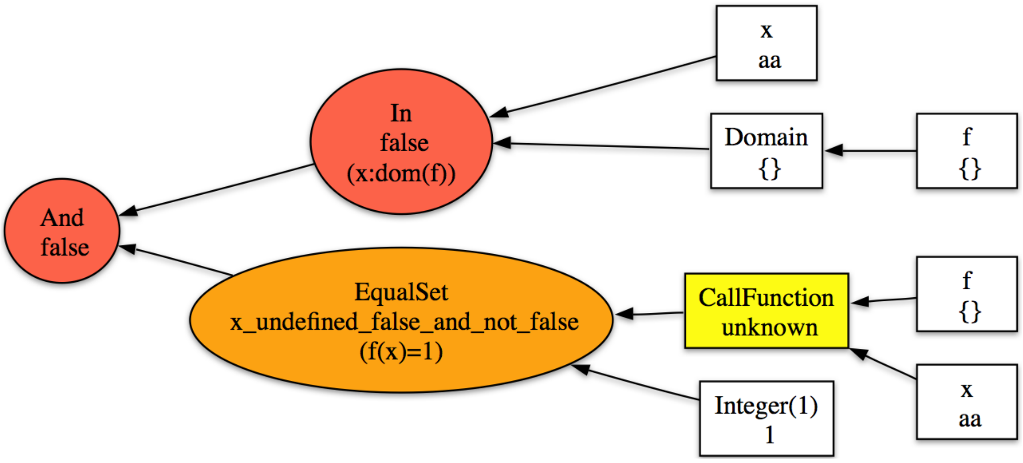


Fig. 5. Visualising a predicate with undefined sub-predicates

One interesting problem is what to do when an existentially quantified formula is false. In this case, our algorithm removes conjuncts from the end of the body of the quantified formula, until the formula becomes true. An example is shown in Figure 6, where we apply our algorithm to the guard of the `new` operation of the scheduler machine (for the state already depicted earlier in Figure 3). Note that the guard is modelled by an existential quantification over the parameters of the operation. As can be seen, the existential formula is false, but our algorithm has detected that by removing the first condition $pp \notin active$, the formula would become satisfiable. This information can be very valuable in detecting exactly where an inconsistency arises inside a formula. In previous works [5,11], different algorithms based on SAT solvers were developed to find a Minimal Unsatisfiable Subformula (MUS). Our approach to find a short unsatisfiable formula is simple. We are currently interested in improving our algorithm by evaluating how the approaches in [5,11] can be applied to our Prolog constraint solving technique.

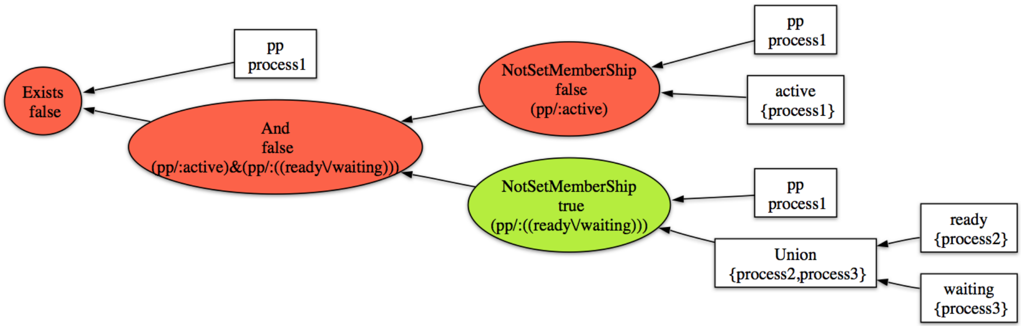


Fig. 6. Visualising the guard of `new` of the scheduler for the state of Figure 3

3 The new Graphical Animation Model

The animation model is very simple:

(i) The basic units are individual images. The images are given a number and their source file location is declared in the DEFINITIONS section of the animated machine. A definition `ANIMATION_IMG`x `==` "*filename*", defines the image with number x where *filename* is the path to a gif image file.

(ii) The graphical visualisation consists of a two-dimensional grid, each cell in the grid can contain an image. The same image can appear multiple times in the grid.

(iii) The graphical visualisation is recomputed for every state, by evaluating a user-defined animation function f_a. The animation function f_a is declared by defining `ANIMATION_FUNCTION` in the DEFINITIONS section and must be of type `INTEGER * INTEGER +-> INTEGER`. If the function is defined for r and c, this means that the animator should display the image with number $f_a(r, c)$ at row r and column c. If f_a is undefined at r and c, then no image is displayed in that cell.

The dimension of the grid is computed by looking at the minimum and maximum coordinates that occur in the animation function. More precisely, the rows are in the range $min(dom(dom(f_a)))..max(dom(dom(f_a)))$ and the columns are in the range $min(ran(dom(f_a)))..max(ran(dom(f_a)))$.

For our 8-puzzle example, one could thus write the following animation function, together with an image declaration list. The result of using this animation function can be seen on the right of Figure 1.

```
ANIMATION_FUNCTION == ( {r,c,i|r:1..dim & c:1..dim & i=0} <+ board);
ANIMATION_IMG0 == "images/sm_empty_box.gif";  /* empty square */
```

```
ANIMATION_IMG1 == "images/sm_1.gif";  /* square with 1 inside */
ANIMATION_IMG2 == "images/sm_2.gif";
ANIMATION_IMG3 == "images/sm_3.gif";
ANIMATION_IMG4 == "images/sm_4.gif";
ANIMATION_IMG5 == "images/sm_5.gif";
ANIMATION_IMG6 == "images/sm_6.gif";
ANIMATION_IMG7 == "images/sm_7.gif";
ANIMATION_IMG8 == "images/sm_8.gif"; /* square with lightblue 8 inside */
```

Each of the three integer types in the signature can be replaced by a deferred or enumerated set, in which case our tool translates elements of this set into numbers. In case of enumerated sets, the number is position of the element in the definition of the set in the SETS clause. Deferred set elements are numbered internally by ProB, and this number is used. (Note, however, that the whole animation function has to be of the same type; otherwise the animator will complain about a type error.)

To avoid having to produce images for simple strings, one can use a declaration `ANIMATION_STR`x `== "my string"` to define image with number x to be automatically generated from the given string.

Typical patterns for the animation function are as follows:

- A useful way to obtain a function of the required signature is to write a set comprehension of the following form:
  `{row,col,img | row:1..NrRow & col:1..NrCols & P}`, where `P` is a predicate which gives `img` a value depending on `row` and `col`.
- Another useful pattern is to write one function for default images, and then use the override operator to replace the default images only when needed: `DefaultImages <+ CurrentImages`. This results in much more concise and readable functions. This was used in the 8-Puzzle, by setting as default the empty square (image 0) overriden by the partially defined `board` function.
- Translation predicates between user sets and numbers (extension above can directly handle user sets, but does not work well if we need a special image for undefined,...)

4 Further Examples

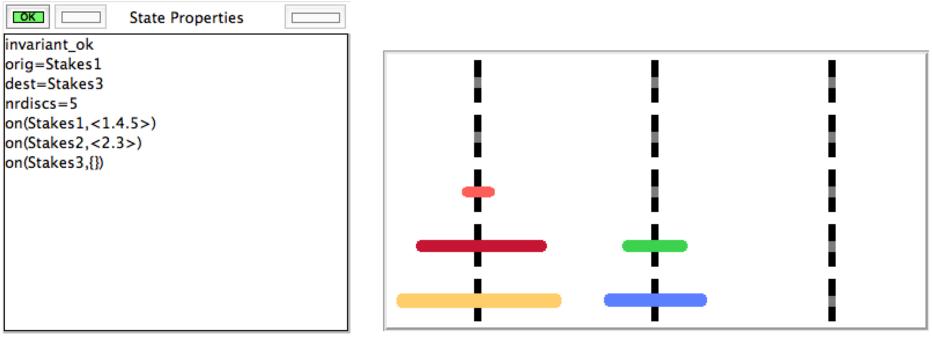
Below we show three more examples, which illustrate how our new animation model can be used. The examples also show that, despite its simplicity, the model is powerful enough to provide interesting animations for a variety of models. This will be further corroborated in Section 5.

4.1 Towers of Hanoi

The Towers of Hanoi problem is widely used to teach recursion and problem solving. A B model of this problem is as follows. The animation function is surprisingly simple (although we needed to make it slightly more complicated to ensure that the stakes are not shown upside down). The graphical result can be seen in Figure 7.

```
MACHINE Hanoi
SETS Stakes
DEFINITIONS GOAL == (!s.(s:Stakes & s/=dest => on(s) = <>));
scope_Stakes == 1..3;
ANIMATION_FUNCTION == ({r,c,i|r:1..nrdiscs & c:Stakes & i=0} <+
  {r,c,i|r:1..nrdiscs &
   c:Stakes & r-5+size(on(c)): dom(on(c)) &
   i = on(c)(r-5+size(on(c)))});
```

```
ANIMATION_IMG0 == "images/Disc_empty.gif";
ANIMATION_IMG1 == "images/Disc1.gif";   /* the smallest disc */
ANIMATION_IMG2 == "images/Disc2.gif";
ANIMATION_IMG3 == "images/Disc3.gif";
ANIMATION_IMG4 == "images/Disc4.gif";
ANIMATION_IMG5 == "images/Disc5.gif";  /* the largest disc */
CONSTANTS orig,dest,nrdiscs
PROPERTIES orig: Stakes & dest:Stakes & orig /= dest & nrdiscs = 5
VARIABLES on
INVARIANT on : Stakes --> seq(INTEGER)
INITIALISATION
   on := %s.(s:Stakes & s /= orig | <>) \/
   {orig |-> %x.(x:1..nrdiscs|x)}
OPERATIONS
   Move(from,to,disc) =
     PRE
      from:Stakes & on(from) /= <> & to:Stakes & to /= from &
      disc:NATURAL1 & disc = first(on(from)) &
      (on(to) /= <> =>  first(on(to))> disc)
     THEN
      on := on <+ { from |-> tail(on(from)), to |-> (disc -> on(to))}
   END
END
```



invariant_ok
orig=Stakes1
dest=Stakes3
nrdiscs=5
on(Stakes1,<1.4.5>)
on(Stakes2,<2.3>)
on(Stakes3,[])

Fig. 7. Hanoi Non-Graphical and Graphical Visualisation

4.2 Scheduler

We return to the scheduler example from [8], whose code is in Figure 2. Here we require a more complicated animation function, because we have to map `PID` elements to image numbers. The result of the animation has already been shown in Figure 3.

```
IsPidNrci ==
 ((p=process1 & i=1) or (p=process2 & i=2) or (p=process3 & i=3));
ANIMATION_FUNCTION ==  ({1|->0|->5, 2|->0|->6, 3|->0|->7} \/
  {r,c,img|r:1..3 & img=4 & c:1..3} <+
  ({r,c,i| r=1 & i:INTEGER & c=i & #p.(p:waiting & IsPidNrci)} \/
   {r,c,i| r=2 & i:INTEGER & c=i & #p.(p:ready & IsPidNrci)} \/
   {r,c,i| r=3 & i:INTEGER & c=i & #p.(p:active & IsPidNrci)} ));
ANIMATION_IMG1 == "images/1.gif";
ANIMATION_IMG2 == "images/2.gif";
ANIMATION_IMG3 == "images/3.gif";
ANIMATION_IMG4 == "images/empty_box.gif";
ANIMATION_IMG5 == "images/Waiting.gif";
ANIMATION_IMG6 == "images/Ready.gif";
ANIMATION_IMG7 == "images/Active.gif"
```

It would have been more elegant to use subsidiary definitions with arguments, such as:

```
IsPidNr(c,i) ==
 ((c=process1 & i=1) or (c=process2 & i=2) or (c=process3 & i=3))
```

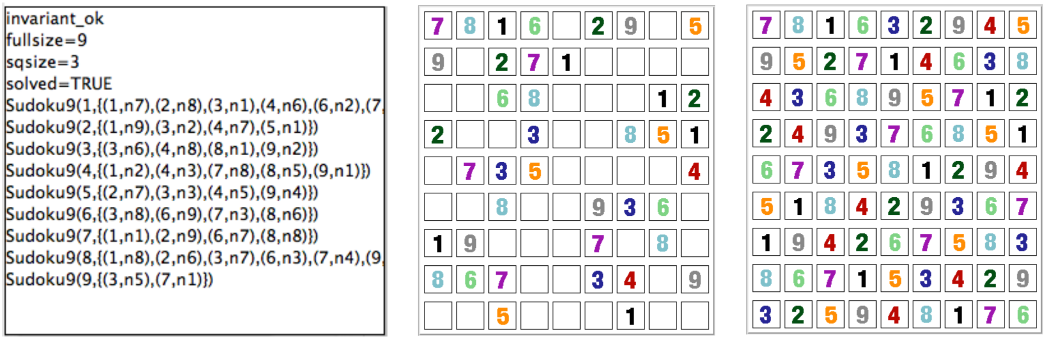
However, the current parser of ProB (derived from jbtools) cannot deal with definitions with arguments when used inside other definitions. We are currently deploying a new parser developed by Fabian Fritz using SableCC, which will overcome this problem.

4.3 Sudoku

For our course, we have also developed a B model of Sudoku, and show students how they can use ProB to solve Sudoku puzzles. The machine has the variable `Sudoku9` of type `1..fullsize-->(1..fullsize+->NRS)`, where `NRS` is an enumerate set `{n1, n2, ...}` of cardinality `fullsize`. The animation function is as follows:

```
Nri == ((Sudoku9(r)(c)=n1 => i=1) & (Sudoku9(r)(c)=n2 => i=2) &
        (Sudoku9(r)(c)=n3 => i=3) & (Sudoku9(r)(c)=n4 => i=4) &
        (Sudoku9(r)(c)=n5 => i=5) & (Sudoku9(r)(c)=n6 => i=6) &
        (Sudoku9(r)(c)=n7 => i=7) & (Sudoku9(r)(c)=n8 => i=8) &
        (Sudoku9(r)(c)=n9 => i=9)  );
ANIMATION_FUNCTION == ( {r,c,i|r:1..fullsize & c:1..fullsize & i=0} <+
    {r,c,i|r:1..fullsize & c:1..fullsize & c:dom(Sudoku9(r)) &
           i:1.. fullsize & Nri} );
```

Figure 8 shows the non-graphical visualisation of a particular puzzle, then the graphical visualisation of the puzzle as well as the visualisation of the solution found by ProB (after a couple of seconds).



```
invariant_ok
fullsize=9
sqsize=3
solved=TRUE
Sudoku9(1,{(1,n7),(2,n8),(3,n1),(4,n6),(6,n2),(7,
Sudoku9(2,{(1,n9),(3,n2),(4,n7),(5,n1)})
Sudoku9(3,{(3,n6),(4,n8),(8,n1),(9,n2)})
Sudoku9(4,{(1,n2),(4,n3),(7,n8),(8,n5),(9,n1)})
Sudoku9(5,{(2,n7),(3,n3),(4,n5),(9,n4)})
Sudoku9(6,{(3,n8),(6,n9),(7,n3),(8,n6)})
Sudoku9(7,{(1,n1),(2,n9),(6,n7),(8,n8)})
Sudoku9(8,{(1,n8),(2,n6),(3,n7),(6,n3),(7,n4),(9,
Sudoku9(9,{(3,n5),(7,n1)})
```

7	8	1	6		2	9		5
9		2	7	1				
		6	8				1	2
2			3			8	5	1
	7	3	5					4
		8			9	3	6	
1	9				7		8	
8	6	7			3	4		9
		5				1		

7	8	1	6	3	2	9	4	5
9	5	2	7	1	4	6	3	8
4	3	6	8	9	5	7	1	2
2	4	9	3	7	6	8	5	1
6	7	3	5	8	1	2	9	4
5	1	8	4	2	9	3	6	7
1	9	4	2	6	7	5	8	3
8	6	7	1	5	3	4	2	9
3	2	5	9	4	8	1	7	6

Fig. 8. Sudoku Non-Graphical and Graphical Visualisation

Note that it would have been nice to be able to replace `Nri` inside the animation function simply by `i = Sudoku9(r)(c)`. While our visualisation algorithm can automatically convert set elements to numbers, the problem is that there is a type error in the override: the left-hand side is a function of type `INTEGER*INTEGER+->INTEGER` while the right-hand side now becomes a function of type `INTEGER*INTEGER+->NRS`. One solution is to extend our animator to accept multiple definitions of the animation function. We have done so, and the user can, in addition to the standard animation function, optionally define a default background animation function. The standard animation function will override the default animation function, but the overriding is done within the graphical animator and not within a B formula. In this way, one can now rewrite the above animation as follows:

```
ANIMATION_FUNCTION_DEFAULT == ( {r,c,i|r:1..fullsize & c:1..fullsize & i=0} );
ANIMATION_FUNCTION == ( {r,c,i|r:1..fullsize & c:1..fullsize &
    c:dom(Sudoku9(r)) & i:1.. fullsize &  i = Sudoku9(r)(c)} )
```

The scheduler animation from Section 4.2 can now also be rewritten more elegantly as follows:

```
  ANIMATION_FUNCTION_DEFAULT ==
     ( {1|->0|->5, 2|->0|->6, 3|->0|->7} \/ {r,c,img|r:1..3 & img=4 & c:1..3} );
  ANIMATION_FUNCTION == ( {r,c,i| r=1 & i:PID & c=i & i:waiting} \/
                               {r,c,i| r=2 & i:PID & c=i & i:ready} \/
                               {r,c,i| r=3 & i:PID & c=i & i:active}
                              )
```

5 A more detailed case study: Formalising and Visualising a Lift

In this section, we provide a more detailed case study and apply our animation technology to a bigger specification, similar to a typical student project: the model of a lift along with a controller that ensures that requests are eventually served.

Our experience was very positive, it was relatively straightforward to provide an appealing graphical visualisation of the model. This greatly enhanced the understandability of the model, and allowed us to spot errors more quickly (e.g., in one version of the model, the lift could get stuck on the top floor, which was not that obvious to spot in the non-graphical visualisation).

In the following, we actually present two models of a lift. The first model makes no distinction between internal and external call buttons. The second model is a refinement of the first, and here there are separate panel buttons inside the lift and external call buttons on each floor. In both cases, the lift can move between a ground floor (constant `groundf`) and a top floor (constant `topf`). The state of both lift models consists of the current floor the lift is on (variable `cur_floor`), whether its door is open or not (variable `door_open`), whether it is currently moving up or down (variable `direction_up`) as well as the state of the buttons. In Figure 9, we present the graphics we used in the definition `ANIMATION_IMG`x `==` `"`*filename*`"` and describe their associated event. Note that the images used before refinement are from x=0 to 6.

5.1 The Lift Model before Refinement

Below are the constants and variables of the first lift model.

```
MODEL LiftM
CONSTANTS groundf,topf
PROPERTIES
   topf : INTEGER &
   groundf : INTEGER &
   groundf = -1 &
   topf = 2 &
   groundf < topf
VARIABLES  call_buttons,cur_floor,direction_up,do_count,
   door_open,inside_panel_buttons
INVARIANT
   cur_floor : groundf .. topf &
   door_open : BOOL &
   call_buttons : POW(groundf .. topf) &
   direction_up : BOOL &
   inside_panel_buttons : POW(groundf .. topf) &
   do_count : 0 .. 3
   ...
```

Below is our animation function. The left part of the relational overriding `<+` gives the default values (i.e., images), and the right part gives the actual images.

```
DEFINITIONS
  Rconv == topf-r+groundf;
  ANIMATION_FUNCTION ==
   ( {r,c,i|r:groundf..topf & ((c=2 & i=0) or (c=1 & i=2))} <+
    ({r,c,i|r:groundf..topf & Rconv:call_buttons & c=2 & i=1} \/
     {r,c,i|r:groundf..topf & Rconv=cur_floor & c=1 &
        ((door_open=TRUE & i=3) or (door_open=FALSE & i=4))}) \/
     {r,c,i| r=topf+1 & c=1 & ((direction_up=TRUE & i=5)
        or (direction_up=FALSE & i=6)) } );
```

Our default images are 0 and 2. Our current images are 1, 3, 4, 5 and 6. In our case, the lowest floor `groundf` and the highest floor `topf` are equal to −1

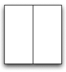
			x	"*filename*"	Image		Description of the Event
`ANIMATION_IMG` *x* == "*filename*"	`After Refinement`	`Before Refinement`	0	`"images/lift/B_CallButtonOff.gif"`	○	Outside the lift	A call button is off.
			1	`"images/lift/B_CallButtonOn.gif"`	●		A call button is on.
			2	`"images/lift/LiftEmpty.gif"`			The lift is not on a specific floor.
			3	`"images/lift/B_LiftOpen.gif"`			The lift's door is opened.
			4	`"images/lift/B_LiftClosed.gif"`			The lift's door is closed.
			5	`"images/lift/B_up_arrow.gif"`	⬆		The lift's direction up is on.
			6	`"images/lift/B_down_arrow.gif"`	⬇		The lift's direction down is off.
			7	`"images/lift/B_up_arrow_off.gif"`	⇧		The lift's direction up is off.
			8	`"images/lift/B_floor_U1_off.gif"`	U1	Inside the lift	The panel button U1 is off.
			9	`"images/lift/B_floor_U1_on.gif"`	U1		The panel button U1 is on.
			10	`"images/lift/B_floor_E_off.gif"`	E		The panel button E is off.
			11	`"images/lift/B_floor_E_on.gif"`	E		The panel button E is on.
			12	`"images/lift/B_floor_1_off.gif"`	1		The panel button 1 is off.
			13	`"images/lift/B_floor_1_on.gif"`	1		The panel button 1 is on.
			14	`"images/lift/B_floor_2_off.gif"`	2		The panel button 2 is off.
			15	`"images/lift/B_floor_2_on.gif"`	2		The panel button 2 is on.
			16	`"images/lift/B_down_arrow_off.gif"`	⇩		The lift's direction down is off.
			17	`"images/lift/B_floor_U1.gif"`	U1	Outside the lift	Floor U1
			18	`"images/lift/B_floor_E.gif"`	E		Floor E
			19	`"images/lift/B_floor_1.gif"`	1		Floor 1
			20	`"images/lift/B_floor_2.gif"`	2		Floor 2

Fig. 9. The Definition `ANIMATION_IMG`x `==` `"`$filename$`"` and the Description of the Event associated to each Image

and 2, respectively. The minimum and maximum coordinates, which occur in the animation function, are 2 and 5, respectively. Then, the dimension of the grid is 3x2. To determine the value of r, we sometimes use `Rconv==topf-r+groundf`. The values of `Rconv` are in the ranges `groundf..topf`. `Rconv` depends on the values obtained, when a call button is pushed or when the lift is on the current floor. For instance, if the current floor `cur_floor` is equal to 2, then `Rconv` is also equal to 2. Consequently, applying `r==topf+groundf-Rconv`, the row r is equal to -1. If `door_open=TRUE`, then the image number `i=3` is displayed at column 1. Otherwise, if `door_open=FALSE`, then the image number `i=4` appears at column 1. More details are presented in Figure 10. Due to space restrictions we do not show the graphical visualisation of this model, only of the more refined model later in Figure 12.

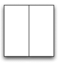
		r	c	i = *x*	Rconv= =topf-r+groundf	Image	Explanation
ANIMATION_FUNCTION == DefaultImages <+ CurrentImages	DefaultImages	groundf..topf	2	0			When a call button is off, the default image is displayed at column 2.
			1	2			When the lift is not on a specific floor, the default image is displayed at column 1.
	CurrentImages	groundf..topf	2	1	Rconv:call_buttons (call_buttons are in the range groundf..topf. Then, Rconv is also in this range. r is computed as r=topf+groundf-Rconv)		A call button is on. It is displayed at column 2. The row is determined according to the value of **Rconv**.
		groundf..topf	1	3	Rconv=cur_floor (The value of the current floor cur_floor is in the range groundf..topf. Then, Rconv is also in this range. r is computed as r=topf+groundf-Rconv)		The image is displayed at column 1, when **door_open=TRUE** (the lift's door is opened).
				4			The image is displayed at column 1, when **door_open=FALSE** **(**the lift's door is closed).
		topf+1	1	5			The image is displayed at column 1 and row 3, when **direction_up=TRUE** (the lift's direction is up).
				6			The image is displayed at column 1 and row 3, when **direction_up=FALSE** (the lift's direction is down).

Fig. 10. The Definition ANIMATION_FUNCTION before Refinement

5.2 The Lift Model after Refinement

After refinement, the ProB animation function is as follows. Note that, due to the distinction between internal buttons and external call buttons, the graphical visualisation has become more sophisticated and the animation function thus more complicated.

```
DEFINITIONS
  pushed_buttons == (call_buttons \/ inside_panel_buttons);
  Rconv == topf-r+groundf;
  ANIMATION_FUNCTION ==
   ({r,c,i|r:groundf..topf & ((c=3 & i=0) or (c=2 & i=2))} <+
    ({r,c,i|r:groundf..topf & Rconv:call_buttons & (c=3 & i=1)} \/
    {r,c,i|r:groundf..topf & Rconv=cur_floor & c=2 &
      ((door_open=TRUE & i=3) or (door_open=FALSE & i=4))}) \/
    {r,c,i|c=1 & ((r=groundf & i=20) or (r=groundf+1 & i=19) or
      (r=topf-1 & i=18) or (r=topf & i=17)) } \/
    {r,c,i| r=topf+1 &
      ((direction_up=TRUE & ((c=1 & i=5) or (c=6 & i=16))) or
      (direction_up=FALSE & ((c=1 & i=7) or (c=6 & i=6)))) } \/
    {r,c,i|r=topf+1 & c=2 & (((-1):inside_panel_buttons & i=9) or
      ((-1)/:inside_panel_buttons & i=8))} \/
    {r,c,i|r=topf+1 & c=3 & ((0:inside_panel_buttons & i=11) or
      ((0/:inside_panel_buttons) & i=10))} \/
    {r,c,i|r=topf+1 & c=4 & ((1:inside_panel_buttons & i=13) or
      (1/:inside_panel_buttons & i=12))} \/
    {r,c,i|r=topf+1 & c=5 & ((2:inside_panel_buttons & i=15) or
      (2/:inside_panel_buttons & i=14))});
...
```

In the lift model LiftR_1, we use 21 gif images. Note that the first seven gif images were also used in the lift model LiftM. In the relational overriding <+, the default values (i.e., images) are 0 and 2 and our current images are 1 and from 3 to 20. As in the lift model LiftM, groundf and topf are equal to -1 and 2, respectively. The dimension of the grid is 3x6. Moreover, we sometimes use Rconv in order to compute the value of r. More details about the definition ANIMATION_FUNCTION; i.e, the position of each image in the graphical visualisation, the image's number and

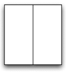
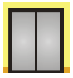
		r	c	i=*x*	Rconv= = topf-r+groundf	Image	Explanation
ANIMATION_FUNCTION == DefaultImages <+ CurrentImages	DefaultImages	groundf..topf	3	0		○	When a call button is off, the default image is displayed at column 3.
			2	2			When the lift is not on a specific floor, the default image is displayed at column 2.
	CurrentImages	groundf..topf	3	1	Rconv:call_buttons (call_buttons are in the range groundf..topf. Then, Rconv is also in this range. r is computed as r=topf+groundf-Rconv)	●	A call button is on. It is displayed at column 3. The row is determined according to the value of **Rconv**.
		groundf..topf	2	3	Rconv=cur_floor (The value of the current floor cur_floor is in the range groundf..topf. Then, Rconv is also in this range. r is computed as r=topf+groundf-Rconv)		The image is displayed at column 2, when **door_open=TRUE** (the lift's door is opened).
				4			The image is displayed at column 2, when **door_open=FALSE** (the lift's door is closed).
		topf+1	1	5		↑	The image is displayed at column 1 and row 3, when **direction_up=TRUE** (the lift's direction is up).
				7		⇧	The image is displayed at column 1 and row 3, when **direction_up=FALSE**
			6	6		↓	The image is displayed at column 6 and row 3, when **direction_up=FALSE** (the lift's direction is down).
				16		⇩	The image is displayed at column 6 and row 3, when **direction_up=TRUE**
		groundf	1	20		2	The image is displayed at column 1 and row -1.
		groundf+1		19		1	The image is displayed at column 1 and row 0.
		topf-1		18		E	The image is displayed at column 1 and row 1.
		topf		17		U1	The image is displayed at column 1 and row 2
		topf+1	2	8		U1	If (-1)/:inside_panel_buttons, the image is displayed at column 2 and row 3.
				9		U1	If (-1):inside_panel_buttons, the image is displayed at column 2 and row 3.
		topf+1	3	10		E	If 0/:inside_panel_buttons, the image is displayed at column 3 and row 3.
				11		E	If 0:inside_panel_buttons, the image is displayed at column 3 and row 3.
		topf+1	4	12		1	If 1/:inside_panel_buttons, the image is displayed at column 4 and row 3.
				13		1	If 1:inside_panel_buttons, the image is displayed at column 4 and row 3.
		topf+1	5	14		2	If 2/:inside_panel_buttons, the image is displayed at column 5 and row 3.
				15		2	If 2:inside_panel_buttons, the image is displayed at column 5 and row 3.

Fig. 11. The Definition ANIMATION_FUNCTION after Refinement

the image's associated event are provided in Figure 11.

Figure 12 contains an animation sequence, starting from the initial state (a1), where the lift is on the ground floor, with its door closed, no buttons pressed and the lift controller being in upwards mode. Between states (a1) and (a2), the operation `push_call_button(-1)` has been executed. The user can clearly see the effect of the operation. This operation now also enables the `open_door` operation, after whose execution we reach state (a3). This enables the `close_door` operation, whose execution leads us to state (a4). Here, we can clearly see that the call button to floor U1 has been turned off again. After (a4), the user has executed the operation `push_call_button(1)` leading to (a5). This enabled the `move_up` operation, after whose execution we reach state (b1), etc...

6 Related Work and Conclusions

As far as related work is concerned, we would like to mention the Possum animation tool [7] for Z. The latter is probably most related, as it allows the user to write custom TclTk code that can query the state of a Z specification in order to provide a custom graphical visualisation. The most closely related work on the B side is [4,1], which uses a special purpose constraint solver over sets (CLPS) to animate B and Z specifications using the so-called BZ-Testing-Tools. However, the focus of these tools is test-case generation and not verification, and the subset of B that is supported is comparatively smaller (e.g., no set comprehensions or lambda abstractions, constants and properties nor multiple machines are supported). To our knowledge no graphical visualisation for states is available.

In summary, we have provided two new graphical visualisation features for ProB: one to visualise predicates as a tree and the other to view the state of a B model in a domain-specific manner. We have shown the usefulness of these features on a series of examples. We hope that this provide further value to teaching B to students.

Acknowledgements

We would like to thank Daniel Plagge for his suggestions in Section 2. We also wish to extend our thanks to the reviewers for their helpful comments.

References

[1] F. Ambert, F. Bouquet, S. Chemin, S. Guenaud, B. Legeard, F. Peureux, M. Utting, and N. Vacelet. BZ-testing-tools: A tool-set for test generation from Z and B using constraint logic programming. In *Proceedings of FATES'02, Formal Approaches to Testing of Software*, pages 105–120, August 2002. Technical Report, INRIA.

[2] AT&T Labs-Research. Graphviz - open source graph drawing software. Obtainable at `http://www.research.att.com/sw/tools/graphviz/`.

[3] J. Bendisposto and M. Leuschel. A generic flash-based animation engine for ProB. In *Proceedings of the 7th International B Conference (B2007)*, LNCS 4355, pages 266–269, Besancon, France, 2007. Springer-Verlag.

[4] F. Bouquet, B. Legeard, and F. Peureux. CLPS-B - a constraint solver for B. In J.-P. Katoen and P. Stevens, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, LNCS 2280, pages 188–204. Springer-Verlag, 2002.

[5] R. Bruni and A. Sassano. Restoring Satisfiability or Maintaining Unsatisfiability by Finding Small Unsatisfiable Subformulae. In H. Kautz and B. Selman, editors, *LICS 2001 Workshop on Theory and Applications of Satisfiability Testing (SAT 2001) Boston (Massachusetts, USA), June 14-15, 2001, Proceedings*. Elsevier Science Pub., 2001.

[6] E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[7] D. Hazel, P. Strooper, and O. Traynor. Requirements engineering and verification using specification animation. *Automated Software Engineering*, 00:302, 1998.

[8] B. Legeard, F. Peureux, and M. Utting. Automated boundary testing from Z and B. In *Proceedings FME'02*, LNCS 2391, pages 21–40. Springer-Verlag, 2002.

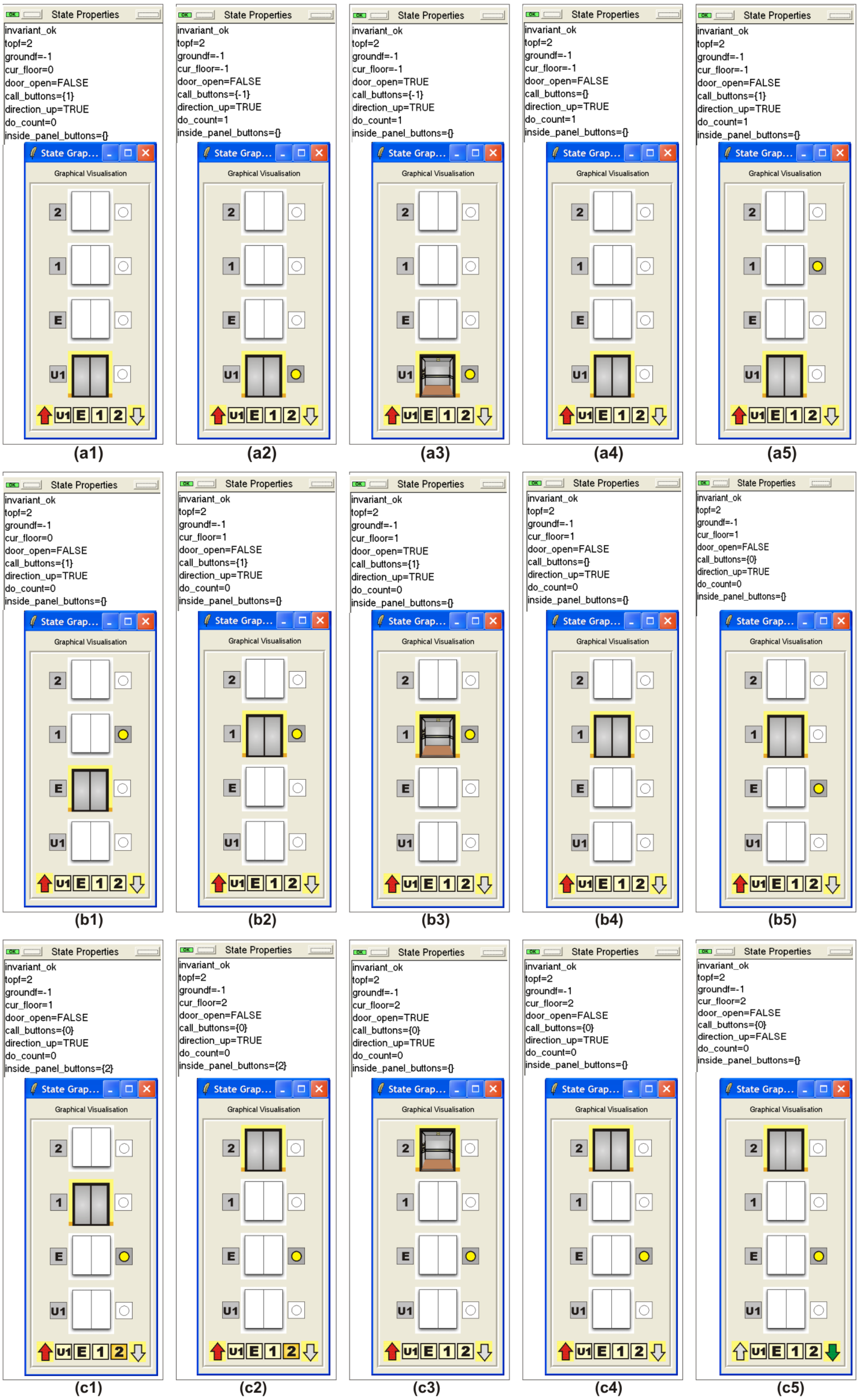
[9] M. Leuschel and M. Butler. ProB: A model checker for B. In K. Araki, S. Gnesi, and D. Mandrioli, editors, *FME 2003: Formal Methods*, LNCS 2805, pages 855–874. Springer-Verlag, 2003.

[10] M. Leuschel and E. Turner. Visualizing larger states spaces in ProB. In H. Treharne, S. King, M. Henson, and S. Schneider, editors, *Proceedings ZB'2005*, LNCS 3455, pages 6–23. Springer-Verlag, April 2005.

[11] E. Torlak, F. S.-H. Chang, and D. Jackson. Finding minimal unsatisfiable cores of declarative specifications. In J. Cuellar, T. Maibaum, and K. Sere, editors, *FM 2008*, volume 5014 of *Lecture Notes in Computer Science*, pages 326–341. Springer, 2008.

A Sudoku Example

```
MACHINE SudokuSETS9
SETS
 NRS = {n1,n2,n3,n4,n5,n6,n7,n8,n9}
DEFINITIONS
  Column(jj) == %ii.(ii:1..fullsize & jj:dom(Sudoku9(ii)) | Sudoku9(ii)(jj));
  ColumnSol(jj) == %ii.(ii:1..fullsize | SudokuSol(ii)(jj));
 SUBSqIndx == {1..3, 4..6, 7..9}
CONSTANTS sqsize,fullsize
PROPERTIES
  fullsize = card(NRS) & sqsize:NAT1 & sqsize*sqsize = fullsize
VARIABLES Sudoku9, solved
INVARIANT
  Sudoku9: 1..fullsize --> (1..fullsize +-> NRS) & solved:BOOL
INITIALISATION
  Sudoku9:= %i1.(i1:1..fullsize|{}) || solved := TRUE
OPERATIONS
  Set(i,j,nr) =
   PRE
    i:1..fullsize & j:1..fullsize & j/: dom(Sudoku9(i)) &
    nr:NRS & nr/:ran(Sudoku9(i)) & nr/:ran(Column(j))
   THEN Sudoku9(i) := Sudoku9(i) \/ {j |-> nr}
  END;
  StartSolve =
   PRE
    solved=TRUE
   THEN solved := FALSE
  END;
  SetPuzzle1 =
   BEGIN
    Sudoku9 := { 1 |-> { 1|->n7, 2|->n8, 3|->n1, 4|->n6,   6|->n2, 7|->n9, 9|->n5 },
                 2 |-> { 1|-> n9, 3|->n2, 4|->n7, 5|->n1 },
                 3 |-> { 3|-> n6, 4|-> n8, 8|->n1, 9|->n2},
                 4 |-> { 1|-> n2, 4|->n3, 7|->n8, 8|->n5, 9|->n1} ,
                 5 |-> { 2|-> n7, 3|->n3, 4|->n5, 9|->n4} ,
                 6 |-> { 3|-> n8, 6|->n9, 7|->n3, 8|->n6}  ,
                 7 |-> { 1|-> n1, 2|->n9, 6|->n7, 8|->n8} ,
                 8 |-> { 1|-> n8, 2|->n6, 3|->n7, 6|->n3, 7|-> n4, 9|-> n9} ,
                 9 |-> { 3|-> n5, 7|->n1} }
   END;
  Solve =
   ANY
    SudokuSol
   WHERE
    solved=FALSE &
    SudokuSol: 1..fullsize --> (1..fullsize --> NRS) & /* all values are specified */
    !(i,j).(i:1..fullsize & j:1..fullsize & j:dom(Sudoku9(i))
     => SudokuSol(i)(j) = Sudoku9(i)(j))& /*all existing values copied from current board*/
    !(i,j1,j2).(i:1..fullsize & j1:1..fullsize & j2:1..fullsize &
    j2 > j1 => (SudokuSol(i)(j1) /= SudokuSol(i)(j2) &  /* all different on a row */
    SudokuSol(j1)(i) /= SudokuSol(j2)(i)    /* all diferent on a column */ )) &
    !(xi,yi).(xi: SUBSqIndx & yi: SUBSqIndx =>
     !(i1,j1,i2,j2).(i1:INTEGER & i2:INTEGER & j1:INTEGER & j2:INTEGER &
    i1:xi & i2:xi & i2 >= i1 & j1:yi & j2:yi &
    (i2=i1 => j2>j1)  /* (i1|->j1) /= (i2|->j2) */
     => SudokuSol(i1)(j1) /= SudokuSol(i2)(j2))
   THEN Sudoku9 := SudokuSol || solved := TRUE
   END
END
```



```
invariant_ok
topf=2
groundf=-1
cur_floor=0
door_open=FALSE
call_buttons={1}
direction_up=TRUE
do_count=0
inside_panel_buttons={}
```

(a1)

```
invariant_ok
topf=2
groundf=-1
cur_floor=-1
door_open=FALSE
call_buttons={-1}
direction_up=TRUE
do_count=1
inside_panel_buttons={}
```

(a2)

```
invariant_ok
topf=2
groundf=-1
cur_floor=-1
door_open=TRUE
call_buttons={-1}
direction_up=TRUE
do_count=1
inside_panel_buttons={}
```

(a3)

```
invariant_ok
topf=2
groundf=-1
cur_floor=-1
door_open=FALSE
call_buttons={}
direction_up=TRUE
do_count=1
inside_panel_buttons={}
```

(a4)

```
invariant_ok
topf=2
groundf=-1
cur_floor=-1
door_open=FALSE
call_buttons={1}
direction_up=TRUE
do_count=1
inside_panel_buttons={}
```

(a5)

```
invariant_ok
topf=2
groundf=-1
cur_floor=0
door_open=FALSE
call_buttons={1}
direction_up=TRUE
do_count=0
inside_panel_buttons={}
```

(b1)

```
invariant_ok
topf=2
groundf=-1
cur_floor=1
door_open=FALSE
call_buttons={1}
direction_up=TRUE
do_count=0
inside_panel_buttons={}
```

(b2)

```
invariant_ok
topf=2
groundf=-1
cur_floor=1
door_open=TRUE
call_buttons={1}
direction_up=TRUE
do_count=0
inside_panel_buttons={}
```

(b3)

```
invariant_ok
topf=2
groundf=-1
cur_floor=1
door_open=FALSE
call_buttons={}
direction_up=TRUE
do_count=0
inside_panel_buttons={}
```

(b4)

```
invariant_ok
topf=2
groundf=-1
cur_floor=1
door_open=FALSE
call_buttons={0}
direction_up=TRUE
do_count=0
inside_panel_buttons={}
```

(b5)

```
invariant_ok
topf=2
groundf=-1
cur_floor=1
door_open=FALSE
call_buttons={0}
direction_up=TRUE
do_count=0
inside_panel_buttons={2}
```

(c1)

```
invariant_ok
topf=2
groundf=-1
cur_floor=2
door_open=FALSE
call_buttons={0}
direction_up=TRUE
do_count=0
inside_panel_buttons={2}
```

(c2)

```
invariant_ok
topf=2
groundf=-1
cur_floor=2
door_open=TRUE
call_buttons={0}
direction_up=TRUE
do_count=0
inside_panel_buttons={}
```

(c3)

```
invariant_ok
topf=2
groundf=-1
cur_floor=2
door_open=FALSE
call_buttons={0}
direction_up=TRUE
do_count=0
inside_panel_buttons={}
```

(c4)

```
invariant_ok
topf=2
groundf=-1
cur_floor=2
door_open=FALSE
call_buttons={0}
direction_up=FALSE
do_count=0
inside_panel_buttons={}
```

(c5)

Fig. 12. Lift Graphical Visualisation