

Master ESSV

Formal Design

Alain Griffault



Year 2011-2012



- 1 The ARC tool
- 2 First manipulation : basics of ALTARICA
- 3 Second manipulation : validation with a model checker
- 4 Formal design of a lift
- 5 Controller synthesis of a tank



- 1 The ARC tool
- 2 First manipulation : basics of ALTARICA
- 3 Second manipulation : validation with a model checker
- 4 Formal design of a lift
- 5 Controller synthesis of a tank

ARC : The ALTARICA Checker

- Infos at <http://altarica.labri.fr/>
- You have to modify your environment (`.bashrc_export`).
- PATH=\$PATH:/net/autre/LABRI/griffaul/bin/Linux/bin
- \$ arc is a command interpreter.

ARC : few commands

- `arc>help` : is a very usefull command.
- `arc>load` : to read a model or a specification.
- `arc>list` : to display objects known by ARC.
- `arc>flatten` : to compute a node's semantic as a leaf.
- `arc>run` : to simulate an ALTARICA node.
- `arc>sequences` : to generate scenarii.
- `arc>exit` : to quit the ALTARICA checker.

ARC : classical usage

- You have to describe your model in a file (.alt) and load it.
- You have to describe requirements in a file (.spe) and load it.
- You have to understand results.

- 1 The ARC tool
- 2 First manipulation : basics of ALTARICA**
- 3 Second manipulation : validation with a model checker
- 4 Formal design of a lift
- 5 Controller synthesis of a tank



First manipulation : basics of ALTARICA

Models

Refer to lesson's slides for the syntax.

- Minimal and FIFO nodes
- Electrical circuit (V1, V2 and corrections).
- Scheduler with and without priority.
- Courses with and without broadcast.

Test various commands such as :

- `$ help, load, list,`
- `$ run` to simulate a node.



First manipulation : basics of ALTARICA

Requirements : first example

```
with nodename [, nodename]* do
  quot() > '$NODENAME.dot';
  show(all) > '$NODENAME.res';
done
```

- \$ more nodename.dot
- \$ dot -Tpdf nodename.dot > nodename.pdf



- 1 The ARC tool
- 2 First manipulation : basics of ALTARICA
- 3 Second manipulation : validation with a model checker
- 4 Formal design of a lift
- 5 Controller synthesis of a tank



Second manipulation : validation with a model checker

The ALTARICA checker ARC

- ARC is a very powerful model-checker for ALTARICA.
- Users can choose to encode models as graphs or as BDD. The first one permits that all properties can be computed in a linear time in the size of the graph, and the second one permits to deal with very big systems.
- To prove that $M \models P$, you have to compute counter examples for P .
 - `notP := any - P;`
 - `notP := formula-describing-P-counter-examples;`and you have to check the result with `test(notP, 0)`.



Second manipulation : validation with a model checker

To do with ARC

You must validate all *ALTARICA* nodes for all examples.

- Compute deadlock and notSCC properties.
- Check for properties and output results in files.
- For each properties witch is not satisfy, compute a counter example and output it in dot format.
- If the number of configurations is not so big, output in dot format the reachability graph.
- Output in files property's cardinals.

You may also compute properties depending of the system's type.

- Electrical circuit : no loop of reactions.
- Scheduler : the priority between pools of jobs is respected.
- Courses : 3 students can't write at the same time.



- 1 The ARC tool
- 2 First manipulation : basics of ALTARICA
- 3 Second manipulation : validation with a model checker
- 4 Formal design of a lift
- 5 Controller synthesis of a tank

Informal description

The lift must be use in any building. Its design must no be dependant on the number of floor.

- At each floor, you may call the lift with a button.
- In the lift, there are as many buttons than floors.
- A lighting button means that this request is not yet satisfy.
- When the lift stops, doors open automatically.

At each time, a software controller chooses the next thing to do between : open a door, close a door, go up, go down or nothing.

The owner of the building wants that these requirements have been proved.

Requirements

- 1 When a button is push, it lights.
- 2 When the corresponding service is done, it lights off.
- 3 At each floor, the door is close if the lift is not here.
- 4 Each request must be honored a day.
- 5 The software opens the door at some floor only if there is some requests for that floor.
- 6 If there is no request, the lift must stay at the same floor.
- 7 When the lift moves, it must stop where there is a request.
- 8 When there are several requests, the software must (if necessary) continue in the same direction than its last move.



How to modelize ?

Remarks

- With finite model-checking we can't prove a property with parameters. For that, we need theorem proving method. So we need to fix the number of floors.
- 1000 seems a good choice since no building in the world have so much floors, but no model checker in the world can deal with such model.
- On the opposite, every model checker can deal with a building with only one floor, but a lift is not usefull in such a building.
- In addition, most of the properties are tautology for a one floor building.



How to modelize ?

The minimal number of floors

No requirements must be a tautology in the model. This means that we have to choose the least number for which any requirement is not trivially satisfied.

- 1 One floor is mandatory.
- 2 One floor is mandatory.
- 3 Two floors are mandatory.
- 4 Three floors are mandatory.
- 5 Two floors are mandatory.
- 6 Two floors are mandatory.
- 7 Three floors are mandatory.
- 8 Three (or four ?) floors are mandatory.

We choose four floors to have more confidence.



How to modelize ?

Open or close system

- An open system is a system with free inputs representing the environment's information. This type of system is use when the environment is not well described and when we want to know in which kind of environment, the system is correct.
- A close system is an open system and its environment describe as a particular component of the whole system.

Users can only push button in this system. The better way to describe users is to abstract them by the push action on button.



How to modelize ?

Architecture or fonctionnal design ?

- ALTAIRICA language is enough general for the two.
- We have to convince the owner of the building. He is certainly not an engineer, nor a computer scientist.
- I think it is easier to convince him with an architecture model witch is certainly less far to the real system than the fonctionnale one.



How to modelize ?

The system to model

How to modelize ?

The hierarchy of the model

- To convince the owner, the hierarchy must reflect the real building.
- A top-down analyse permits to discover :
 - ① Four floors and a lift.
 - ② A door and four buttons in the lift.
 - ③ A door and a button in each floor.



Task of modelling and validation

What kind of button ?

Numerous choice for a button. Analyze of the required functionalities is necessary :

- A push button including a light and not a switch button.
- A signal to light off the button. Is it always possible to (send/receive) this signal or not ?



Task of modelling and validation

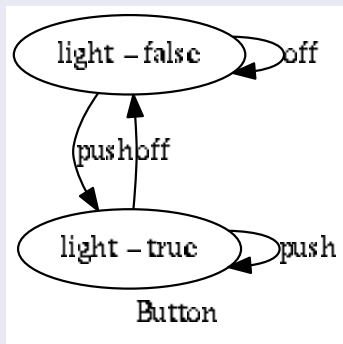
An ALTARICA model of a button

```
/* A Button reacts to
 *   - actions of users
 *   - a signal to light off (even if it is off)
 */
node Button
  state
    light : bool : public;
  init
    light := false;
  event
    push : public;
    off;
  trans
    true | - push -> light := true;
    true | - off -> light := false;
edon
```



Task of modelling and validation

Button's semantic



Task of modelling and validation

What kind of door?

Numerous choice for a door. Analyze of the required functionalities is necessary :

- An unique signal to alternatively open and close the door.
- A signal to close the door (even if the door is close), and another signal to open the door (even if the door is open).



Task of modelling and validation

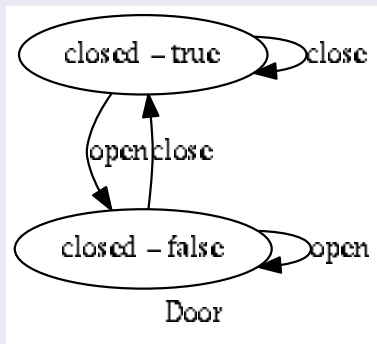
An ALTARICA model of a door

```
/* A Door reacts to:
 *   - a signal to open the door
 *   - a signal to close the door
 */
node Door
  state
    closed : bool : public;
  init
    closed := true;
  event
    open, close : public;
  trans
    true |— open  → closed := false;
    true |— close → closed := true;
edon
```



Task of modelling and validation

Door's semantic



Task of modelling and validation

How to built a floor ?

- A floor contains a button and a door.
- We can send the off signal to the button when the corresponding request is satisfy.
- We have to chose the meaning for “the service is done”
 - The opening instant.
 - The closing instant.



Task of modelling and validation

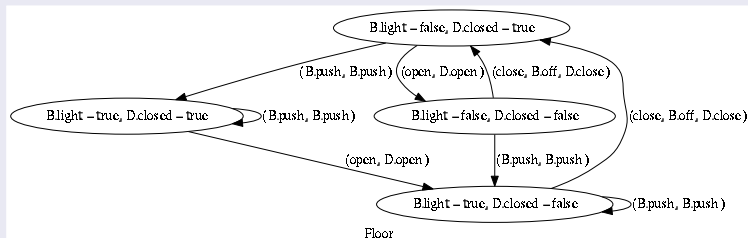
An ALTARICA model of a floor

```
/* A floor is made of a door and a button.  
* We need a meaning for "the service is done"  
*   - it can be the opening instant  
*   - it can be the closing instant  
* We choose the closing instant  
* to send the "off" signal  
*/  
node Floor  
  sub  
    B : Button;  
    D : Door;  
  event  
    close, open;  
  trans  
    ~D.closed |- close -> ;  
    D.closed  |- open  -> ;  
  sync  
    <close, D.close, B.off>;  
    <open, D.open>;  
edon
```



Task of modelling and validation

Floor's semantic



Task of modelling and validation

How to built a lift ?

- A lift contains four buttons and a door.
- A lift moves only if its door is closed.
- We can send the off signal to the appropriate button when the corresponding request is satisfy.
- We have to chose the meaning for “the service is done”.
 - The opening instant.
 - The closing instant.

We made the same choice as for the floor.



Task of modelling and validation

An ALTARICA model of a lift

```
/* A lift contains one button peer floor (4) and a door.  
 * Same choices as for the Floor component.  
 */
```

```
node Lift
```

```
  state   floor : [0,3] : parent;      init   floor := 0;
```

```
  sub     D : Door; B : Button[4];
```

```
  event   up, down, close[4], open;
```

```
  trans   D.closed |— up    → floor := floor + 1;
```

```
          D.closed |— down → floor := floor - 1;
```

```
          ~D.closed & floor = 0 |— close[0] → ;
```

```
          ~D.closed & floor = 1 |— close[1] → ;
```

```
          ~D.closed & floor = 2 |— close[2] → ;
```

```
          ~D.closed & floor = 3 |— close[3] → ;
```

```
          D.closed |— open → ;
```

```
  sync    <close[0], D.close, B[0].off>;
```

```
          <close[1], D.close, B[1].off>;
```

```
          <close[2], D.close, B[2].off>;
```

```
          <close[3], D.close, B[3].off>;
```

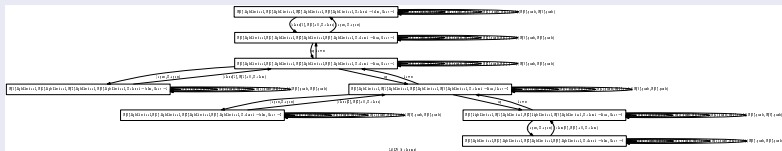
```
          <open, D.open>;
```

```
edon
```



Task of modelling and validation

Lift's semantic



Task of modelling and validation

Lift : the validate command

basic properties checking for node 'Lift'

there is 128 configurations.

usage of variables

All variables are referenced at least once in assertions or transitions.

uniqueness of initial configuration

The system has only one initial configuration

coverage of domains / configurations

any assignment is a configuration.

coverage of domains / reachables

any assignment is reachable.

usage of macro-transitions

All macro-transitions are triggered.



Task of modelling and validation

Lift's validation

```
/*  
 * Properties for node : Lift  
 * # state properties : 2  
 *  
 * any_s = 128  
 * initial = 1  
 *  
 * # trans properties : 4  
 *  
 * any_t = 864  
 * self = 384  
 * epsilon = 128  
 * self_epsilon = 128  
 */  
TEST(deadlock,0)           [PASSED]  
TEST(notResetable,0)      [PASSED]
```



Task of modelling and validation

How is the building ?

The building contains four floors and one lift.

- The lift's door and a floor's door open and close synchronously.
- open is possible at some floor only if there is some request to that floor.
- The lift move up (resp. down) only if there is an up (resp. down) request.



Task of modelling and validation

An ALTARICA model of a building (1)

```
/* The building contains four floors and one lift.  
 *   - The two doors open and close synchronously.  
 *   - open only if some request to that floor exists.  
 *   - up only if some up request exists.  
 *   - down only if some down request exists.  
 */  
node Building1  
  sub  
    F : Floor[4];  
    L : Lift;  
  flow  
    requestUp, requestDown : bool : private;  
    request : bool[4] : private;
```



Task of modelling and validation

An ALTARICA model of a building (2)

assert

```
request[0] = (L.B[0].light | F[0].B.light);  
request[1] = (L.B[1].light | F[1].B.light);  
request[2] = (L.B[2].light | F[2].B.light);  
request[3] = (L.B[3].light | F[3].B.light);  
requestUp  = ((request[3] & L.floor<3) |  
              (request[2] & L.floor<2) |  
              (request[1] & L.floor<1));  
requestDown = ((request[0] & L.floor>0) |  
              (request[1] & L.floor>1) |  
              (request[2] & L.floor>2));
```

event down, up, open[4];

```
trans (L.floor=0) & request[0] | - open[0] -> ;  
      (L.floor=1) & request[1] | - open[1] -> ;  
      (L.floor=2) & request[2] | - open[2] -> ;  
      (L.floor=3) & request[3] | - open[3] -> ;  
requestDown | - down -> ;  
requestUp   | - up   -> ;
```



Task of modelling and validation

An ALTARICA model of a building (3)

```
sync <up,      L.up>;  
    <down,    L.down>;  
    <open[0], L.open, F[0].open>;  
    <open[1], L.open, F[1].open>;  
    <open[2], L.open, F[2].open>;  
    <open[3], L.open, F[3].open>;  
    <L.close[0], F[0].close>;  
    <L.close[1], F[1].close>;  
    <L.close[2], F[2].close>;  
    <L.close[3], F[3].close>;
```

edon



Task of modelling and validation

Too big to draw the graph.

Building : the validate command

basic properties checking for node 'Building1'

there is 1792 configurations.

usage of variables

All variables are referenced at least once in assertions or transitions.

uniqueness of initial configuration

The system has only one initial configuration

coverage of domains / configurations

Domains of variables are covered by the set of configurations.

coverage of domains / reachables

Domains of variables are covered by the set of reachables.

usage of macro-transitions

All macro-transitions are triggered.



Task of modelling and validation

Building's validation

```
/*  
 * Properties for node : Building1  
 * # state properties : 2  
 *  
 * any_s = 1792  
 * initial = 1  
 *  
 * # trans properties : 4  
 *  
 * any_t = 19032  
 * self = 9216  
 * epsilon = 1792  
 * self_epsilon = 1792  
 */  
TEST(deadlock,0)           [PASSED]  
TEST(notResetable,0)       [PASSED]
```



Task of modelling and validation

Building's specific validation

```
/*  
  * Properties for node : Building1  
  * # state properties : 8  
  *  
  * level0 = 448  
  * level1 = 448  
  * level2 = 448  
  * level3 = 448  
  * open0 = 192  
  * open1 = 192  
  * open2 = 192  
  * open3 = 192  
  *  
  * # trans property : 0  
  *  
*/
```



Verification of safety properties

Property P1 : When a button is push, it lights

```
// Safety properties
with Building1, Building2, Building3, Building4DF,
    Building4NDF, Building5NDF do
  // When a button is push, it lights.
  notP1 := any_s &
    (tgt(label F[0].B.push)=[F[0].B.light] |
     tgt(label L.B[0].push)=[L.B[0].light] |
     tgt(label F[1].B.push)=[F[1].B.light] |
     tgt(label L.B[1].push)=[L.B[1].light] |
     tgt(label F[2].B.push)=[F[2].B.light] |
     tgt(label L.B[2].push)=[L.B[2].light] |
     tgt(label F[3].B.push)=[F[3].B.light] |
     tgt(label L.B[3].push)=[L.B[3].light] );
  test(notP1,0) > '$NODENAME.P1';
  traceP1 := trace(initial,any_t,notP1);
  dot(src(traceP1)|tgt(traceP1), traceP1)
    > '$NODENAME-P1.dot';
done
```



Verification of safety properties

Building1 : Property P1

When a button is push, it lights.

TEST(notP1, 0) [*PASSED*]



Verification of safety properties

Property P2 : P2 : When the corresponding service is done, it lights off.

```
// Safety properties
with Building1, Building2, Building3, Building4DF,
   Building4NDF, Building5NDF do
  // When the corresponding service is done,
  // the button lights off.
  notP2 := any_s &
    (tgt(label F[0].close)&[request[0]] |
     tgt(label F[1].close)&[request[1]] |
     tgt(label F[2].close)&[request[2]] |
     tgt(label F[3].close)&[request[3]]);
  test(notP2,0) > '$NODENAME.P2';
  traceP2 := trace(initial,any_t,notP2);
  dot(src(traceP2)|tgt(traceP2), traceP2)
    > '$NODENAME-P2.dot';
done
```



Verification of safety properties

Building1 : Property P2

P2 : When the corresponding service is done, it lights off.

TEST(notP2, 0) [*PASSED*]



Verification of safety properties

Property P3 : At each floor, the door is close if the lift is not here.

```
// Safety properties
with Building1, Building2, Building3, Building4DF,
    Building4NDF, Building5NDF do
// At each floor, the door is close
// if the lift is not here.
notP3 := any_s &
    ([L.floor!=0] - [F[0].D.closed]) |
    ([L.floor!=1] - [F[1].D.closed]) |
    ([L.floor!=2] - [F[2].D.closed]) |
    ([L.floor!=3] - [F[3].D.closed]));
test(notP3,0) > '$NODENAME.P3';
traceP3 := trace(initial,any_t,notP3);
dot(src(traceP3)|tgt(traceP3), traceP3)
    > '$NODENAME-P3.dot';
done
```



Verification of safety properties

Building1 : Property P3

At each floor, the door is close if the lift is not here.

TEST(*notP3*, 0) [*PASSED*]



Verification of safety properties

Property P5 : The software opens the door at some floor only if there is some requests for that floor.

```
// Safety properties
with Building1, Building2, Building3, Building4DF,
   Building4NDF, Building5NDF do
  // The software opens the door at some floor
  // only if there is some requests for that floor.
  notP5 := any_t &
    ((label F[0].D.open - rsrc([request[0]])) |
     (label F[1].D.open - rsrc([request[1]])) |
     (label F[2].D.open - rsrc([request[2]])) |
     (label F[3].D.open - rsrc([request[3]]))) ;
  test(notP5,0) > '$NODENAME.P5';
  traceP5 := trace(initial,any_t,src(notP5));
  ceP5 := reach(src(traceP5),traceP5|notP5);
  dot(ceP5, (traceP5|notP5)) > '$NODENAME-P5.dot';
done
```



Verification of safety properties

Building1 : Property P5

The software opens the door at some floor only if there is some requests for that floor.

TEST(*notP5*, 0) [*PASSED*]



Verification of safety properties

Property P6 : If there is no request, the lift must stay at the same floor.

```
// Safety properties
with Building1, Building2, Building3, Building4DF,
    Building4NDF, Building5NDF do
// If there is no request,
// the lift must stay at the same floor.
notP6 := any_t &
    ((label L.up | label L.down) -
        rsrc([request[0] | request[1] | request[2] | request[3]]));
test(notP6, 0) > '$NODENAME.P6';
traceP6 := trace(initial, any_t, src(notP6));
ceP6 := reach(src(traceP6), traceP6 | notP6);
dot(ceP6, traceP6 | notP6) > '$NODENAME-P6.dot';
done
```



Verification of safety properties

Building1 : Property P6

If there is no request, the lift must stay at the same floor.

TEST(notP6, 0) [*PASSED*]



Verification of safety properties

Property P7 : When the lift moves, it must stop where there is a request.

```
// Safety properties
with Building1, Building2, Building3, Building4DF,
   Building4NDF, Building5NDF do
  // When the lift moves,
  // it must stop where there is a request.
  notP7 := any_t & (label L.up | label L.down) &
               rsrc([L.floor=0 & request[0]] |
                    [L.floor=1 & request[1]] |
                    [L.floor=2 & request[2]] |
                    [L.floor=3 & request[3]]) ;
  test(notP7,0) > '$NODENAME.P7';
  traceP7 := trace(initial,any_t,src(notP7));
  ceP7 := reach(src(traceP7),traceP7|notP7);
  dot(ceP7, (traceP7|notP7)) > '$NODENAME-P7.dot';
done
```



Verification of safety properties

Building1 : Property P7

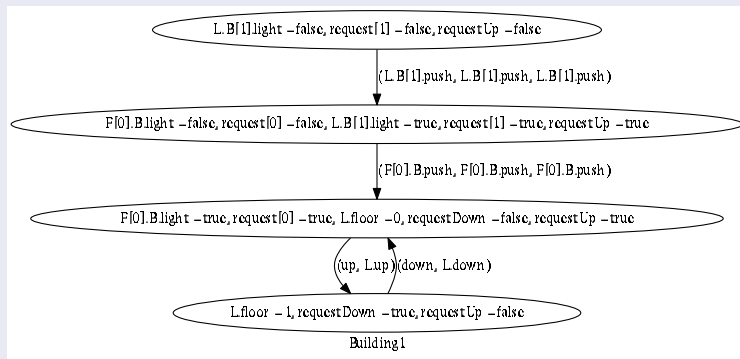
When the lift moves, it must stop where there is a request.

```
TEST(notP7,0)    [FAILED] actual size = 1026
```



Verification of safety properties

Building1 : a counter example for P7.



Verification of safety properties

Building2 : a correction for P7.

```
d7 1
a7 3
/* - the lift moves if no request for the current floor.
*/
node Building2
d25 1
a25 1
  event {down, up} < {open[4]};
```



Verification of safety properties

Building2 : Properties P1, P2, P3, P5, P6 and P7

TEST(notP1, 0) [PASSED]

TEST(notP2, 0) [PASSED]

TEST(notP3, 0) [PASSED]

TEST(notP5, 0) [PASSED]

TEST(notP6, 0) [PASSED]

TEST(notP7, 0) [PASSED]



Verification of safety properties

Property P8 : When there are several requests, the software must (if necessary) continue in the same direction than its last move.

```
// Safety properties
with Building2, Building3, Building4DF, Building4NDF,
    Building5NDF do
  // When there are several requests,
  // the software must (if necessary) continue
  // in the same direction than its last move.
  notP8 := any_t &
    (label L.up & rsrc(src(label L.down)) |
     label L.down & rsrc(src(label L.up))) ;
  test(notP8,0) > '$NODENAME.P8';
  traceP8 := trace(initial,any_t,src(notP8));
  ceP8 := reach(src(traceP8),traceP8|notP8);
  dot(ceP8, (traceP8|notP8)) > '$NODENAME-P8.dot';
done
```



Verification of safety properties

Building2 : Property P8

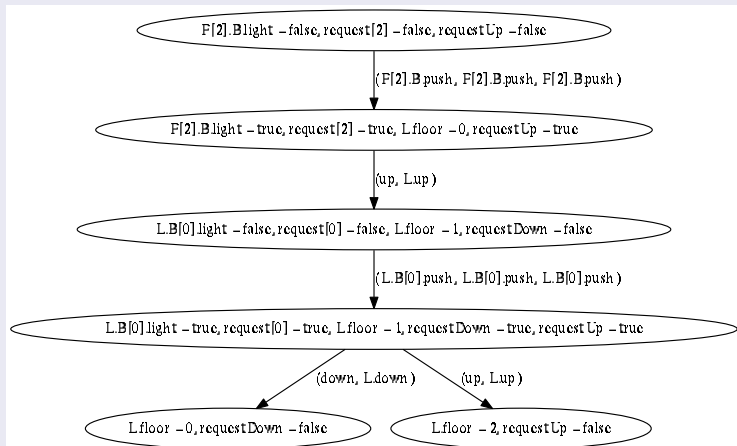
When there are several requests, the software must (if necessary) continue in the same direction than its last move.

```
TEST(notP8, 0)    [FAILED] actual size = 180
```



Verification of safety properties

Building2 : a counter example for P8.



Building2



Verification of safety properties

Building3 : a correction for P8.

```
d9 1
a9 4
/*  - last move of the lift is record in a variable.
   *  - this variable is use to control moves
*/
node Building3
a27 1
  state climb : bool; init climb := false;
d32 2
a33 4
  climb & requestUp           |- up    -> ;
  ~climb & requestDown        |- down  -> ;
  ~climb&~requestDown&requestUp |- up    -> climb:=true;
  climb&~requestUp&requestDown |- down  -> climb:=false;
```



Verification of safety properties

Building3 : Properties P1, P2, P3, P5, P6, P7 and P8

TEST(notP1, 0) [PASSED]

TEST(notP2, 0) [PASSED]

TEST(notP3, 0) [PASSED]

TEST(notP5, 0) [PASSED]

TEST(notP6, 0) [PASSED]

TEST(notP7, 0) [PASSED]

TEST(notP8, 0) [PASSED]



Verification of liveness properties

Property P4 : Each request must be honored a day.
Auxilliary properties

```
with Building3, Building4DF, Building4NDF, Building5NDF do
  exhaustively
    // Preliminary properties for P4
    // we remove "self" to don't
    // take account redundancy "push" events
    waitB0 := any_t&rsrc([L.B[0].light])&rtgt([L.B[0].light])-self;
    waitB1 := any_t&rsrc([L.B[1].light])&rtgt([L.B[1].light])-self;
    waitB2 := any_t&rsrc([L.B[2].light])&rtgt([L.B[2].light])-self;
    waitB3 := any_t&rsrc([L.B[3].light])&rtgt([L.B[3].light])-self;
    waitF0 := any_t&rsrc([F[0].B.light])&rtgt([F[0].B.light])-self;
    waitF1 := any_t&rsrc([F[1].B.light])&rtgt([F[1].B.light])-self;
    waitF2 := any_t&rsrc([F[2].B.light])&rtgt([F[2].B.light])-self;
    waitF3 := any_t&rsrc([F[3].B.light])&rtgt([F[3].B.light])-self;
done
```



Verification of liveness properties

Property P4 : Each request must be honored a day.

```
// Liveness properties
with Building3 do exhaustively
  // Each request must be honored a day.
  notP4 := loop(any_t, waitB0) |
            loop(any_t, waitB1) |
            loop(any_t, waitB2) |
            loop(any_t, waitB3) |
            loop(any_t, waitF0) |
            loop(any_t, waitF1) |
            loop(any_t, waitF2) |
            loop(any_t, waitF3) ;
  test(notP4, 0) > '$NODENAME.P4';
  traceP4 := trace(initial, any_t, src(notP4));
  ceP4 := reach(src(traceP4), traceP4 | notP4);
  dot(ceP4, (traceP4 | notP4)) > '$NODENAME-P4.dot';
done
```



Verification of liveness properties

Building3 : Property P4

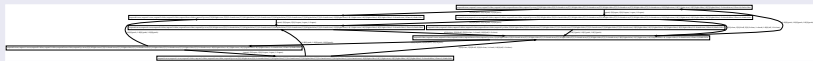
Each request must be honored a day.

```
TEST(notP4, 0)    [FAILED] actual size = 4536
```



Verification of liveness properties

Building3 : a counter example for P4.



Verification of liveness properties

Property P4a : Each request must be honored a day, if the lift moves sometimes.

```
with Building3, Building4DF, Building4NDF, Building5NDF do
  exhaustively
    // A new version of P4: Each request must be
    // honored a day, if the lift moves sometimes.
    move := label L.up | label L.down;
    notP4a := loop(move, waitB0) |
               loop(move, waitB1) |
               loop(move, waitB2) |
               loop(move, waitB3) |
               loop(move, waitF0) |
               loop(move, waitF1) |
               loop(move, waitF2) |
               loop(move, waitF3) ;
    test(notP4a, 0) > '$NODENAME.P4a';
    traceP4a := trace(initial, any_t, src(notP4a));
    ceP4a := reach(src(traceP4a), traceP4a | notP4a);
    dot(ceP4a, traceP4a | notP4a) > '$NODENAME-P4a.dot';
done
```



Verification of liveness properties

Building3 : Property P4a

Each request must be honored a day, if the lift moves sometimes.

TEST(notP4a, 0) [PASSED]



Non determinism and failures

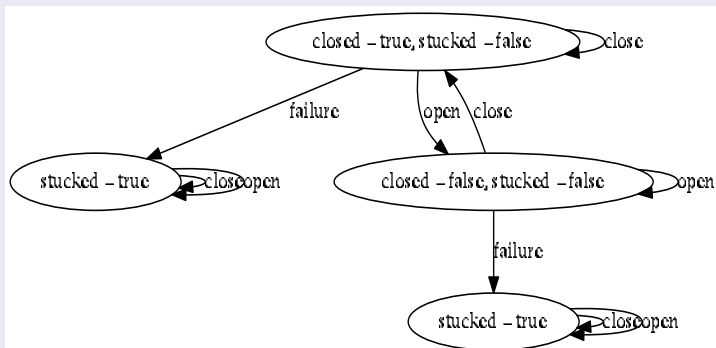
A door with explicit failures

```
/* A Door reacts to:  
*   - a signal to open the door  
*   - a signal to close the door  
* the reaction can lead to a stucked state  
* determinism is used to modelize the failure  
*/  
node DoorDF  
  state closed : bool : public;  
        stucked : bool;  
  event open, close, failure : public;  
  trans not stucked | - open    -> closed := false;  
        not stucked | - close   -> closed := true;  
        not stucked | - failure -> stucked := true;  
        stucked      | - open, close ->;  
  init  closed := true, stucked := false;  
edon
```



Non determinism and failures

A door with explicit failures



Door DF

Non determinism and failures

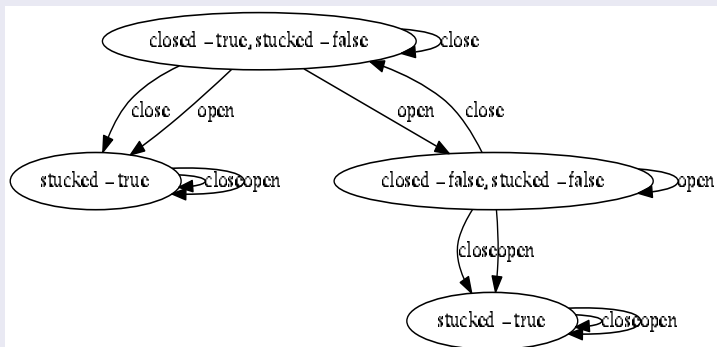
A door with non determinism failures

```
/* A Door reacts to:  
*   - a signal to open the door  
*   - a signal to close the door  
* the reaction can lead to a stucked state  
* non determinism is used to modelize the failure  
*/  
node DoorNDF  
  state closed : bool : public;  
    stucked : bool;  
  event open, close : public;  
  trans not stucked |- open -> closed := false;  
    not stucked |- close -> closed := true;  
    true |- open, close -> stucked := true;  
  init closed := true, stucked := false;  
edon
```



Non determinism and failures

A door with non determinism failures



Door NDF

Non determinism and failures

A floor with explicit failures

```
/* A floor is made of a door and a button.  
 * We need a meaning for "the service is done"  
 * — it can be the opening instant  
 * — it can be the closing instant  
 * We choose the closing instant  
 * to send the "off" signal  
 */
```

```
node FloorDF  
  sub    B : Button; D : DoorDF;  
  event close, open;  
  trans ~D.closed |— close → ;  
        D.closed |— open → ;  
  sync  <close, D.close, B.off>;  
        <close, D.failure, B.off>;  
        <open, D.open>;  
        <open, D.failure>;
```

edon



Non determinism and failures

A floor with non determinism failures

```
/* A floor is made of a door and a button.  
* We need a meaning for "the service is done"  
*   - it can be the opening instant  
*   - it can be the closing instant  
* We choose the closing instant  
* to send the "off" signal  
*/  
node FloorNDF  
  sub   B : Button; D : DoorNDF;  
  event close, open;  
  trans ~D.closed |- close -> ;  
        D.closed |- open -> ;  
  sync  <close, D.close, B.off>;  
        <open, D.open>;  
edon
```



Non determinism and failures

A floor with explicit failures

```
/*  
 * Properties for node : FloorDF  
 * # state properties : 2  
 *  
 * any_s = 8  
 * initial = 1  
 *  
 * # trans properties : 4  
 *  
 * any_t = 28  
 * self = 15  
 * epsilon = 8  
 * self_epsilon = 8  
 */  
TEST(deadlock,0)           [PASSED]  
TEST(notResetable,0)       [FAILED] actual size = 4
```



Non determinism and failures

A floor with non determinism failures

```
/*  
 * Properties for node : FloorNDF  
 * # state properties : 2  
 *  
 * any_s = 8  
 * initial = 1  
 *  
 * # trans properties : 4  
 *  
 * any_t = 28  
 * self = 15  
 * epsilon = 8  
 * self_epsilon = 8  
 */  
TEST(deadlock,0)           [PASSED]  
TEST(notResetable,0)       [FAILED] actual size = 4
```



Non determinism and failures

A lift with explicit failures

```
d4 1
a4 1
node LiftDF
d6 1
a6 1
  sub      D : DoorDF; B : Button[4];
a19 5
    <close[0], D.failure, B[0].off>;
    <close[1], D.failure, B[1].off>;
    <close[2], D.failure, B[2].off>;
    <close[3], D.failure, B[3].off>;
    <open, D.failure>;
```



Non determinism and failures

A lift with non determinism failures

```
d4 1
a4 1
node LiftNDF
d6 1
a6 1
sub      D : DoorNDF; B : Button[4];
```



Non determinism and failures

A building with explicit failures

```
d12 1
a12 1
node Building4DF
d14 2
a15 2
    F : FloorDF[4];
    L : LiftDF;
```



Non determinism and failures

A building with non determinism failures

```
d12 1
a12 1
node Building4NDF
d14 2
a15 2
    F : FloorNDF[4];
    L : LiftNDF;
```



Non determinism and failures

A building with explicit failures

```
/*  
 * Properties for node : Building4DF  
 * # state properties : 2  
 *  
 * any_s = 140952  
 * initial = 1  
 *  
 * # trans properties : 4  
 *  
 * any_t = 1.45551e+06  
 * self = 782142  
 * epsilon = 140952  
 * self_epsilon = 140952  
 */  
TEST(deadlock,0)           [PASSED]  
TEST(notResetable,0)       [FAILED] actual size = 138264
```



Non determinism and failures

A building with non determinism failures

```
/*  
 * Properties for node : Building4NDF  
 * # state properties : 2  
 *  
 * any_s = 140952  
 * initial = 1  
 *  
 * # trans properties : 4  
 *  
 * any_t = 1.45551e+06  
 * self = 782142  
 * epsilon = 140952  
 * self_epsilon = 140952  
 */  
TEST(deadlock,0)           [PASSED]  
TEST(notResetable,0)       [FAILED] actual size = 138264
```



Non determinism and failures

Building4[DF,NDF] : Properties P1, P2, P3, P4a, P5, P6, P7 and P8

`TEST(notP1,0)` [*PASSED*]

`TEST(notP2,0)` [*PASSED*]

`TEST(notP3,0)` [*FAILED*] *actual size = 130200*

`TEST(notP4a,0)` [*FAILED*] *actual size = 90324*

`TEST(notP5,0)` [*PASSED*]

`TEST(notP6,0)` [*PASSED*]

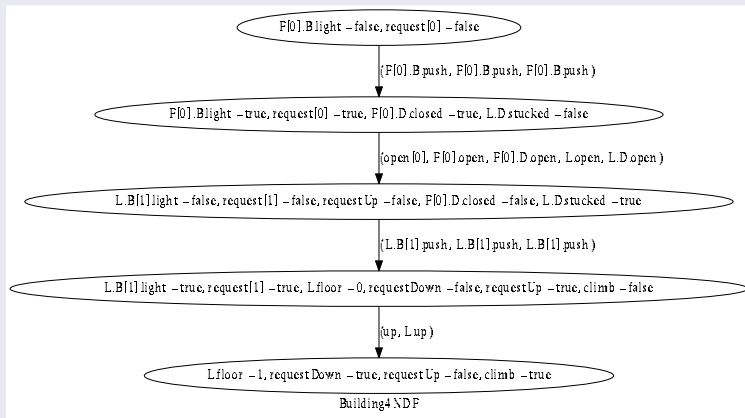
`TEST(notP7,0)` [*FAILED*] *actual size = 43308*

`TEST(notP8,0)` [*PASSED*]



Non determinism and failures

Building4[DF,NDF] : a counter example for P3.



Non determinism and failures

Building5NDF : a correction for P3.

```
d12 1
a12 1
node Building5NDF
a18 1
    doorsAreClosed : bool : private;
a29 4
    doorsAreClosed = ((L.floor=0 & F[0].D.closed) |
                      (L.floor=1 & F[1].D.closed) |
                      (L.floor=2 & F[2].D.closed) |
                      (L.floor=3 & F[3].D.closed));
d36 4
a39 4
    climb & requestUp & doorsAreClosed |- up -> ;
    ~climb & requestDown & doorsAreClosed |- down -> ;
    ~climb & ~requestDown & requestUp & doorsAreClosed |- up ->
        climb:=true;
    climb & ~requestUp & requestDown & doorsAreClosed |- down ->
        climb:=false;
```



Non determinism and failures

Building5NDF : validation

```
/*  
 * Properties for node : Building5NDF  
 * # state properties : 2  
 *  
 * any_s = 10752  
 * initial = 1  
 *  
 * # trans properties : 4  
 *  
 * any_t = 109050  
 * self = 56832  
 * epsilon = 10752  
 * self_epsilon = 10752  
 */  
TEST(deadlock,0)           [PASSED]  
TEST(notResetable,0)       [FAILED] actual size = 8064
```



Non determinism and failures

Building5NDF : Properties P1, P2, P3, P4a, P5, P6, P7 and P8

TEST(notP1, 0) [PASSED]

TEST(notP2, 0) [PASSED]

TEST(notP3, 0) [PASSED]

TEST(notP4a, 0) [PASSED]

TEST(notP5, 0) [PASSED]

TEST(notP6, 0) [PASSED]

TEST(notP7, 0) [PASSED]

TEST(notP8, 0) [PASSED]



The result

- We have to precise some details in the informal description.
 - What is a button and a door?
 - What is the meaning of “The service is done” ?
- We have to precise some requirements.
 - What is the meaning of “Each request must be honored a day” ?
- After that, we have built a model of a lift which satisfy all the requirements.
- At the end, we have shown the power of non determinism to represent failures.

The different tasks

- To obtain a validate small model is not easy.
- To write logical properties is not easy too, but there is a lot of reuse.



Conclusion

Performances

NetBSD amd64 x86_64

4,25	<i>real</i>	3,73	<i>user</i>	0,15	<i>sys</i>
0	<i>maximum resident set size</i>				
0	<i>average shared memory size</i>				
0	<i>average unshared data size</i>				
0	<i>average unshared stack size</i>				
60440	<i>page reclaims</i>				
50	<i>page faults</i>				
0	<i>swaps</i>				
5	<i>block input operations</i>				
133	<i>block output operations</i>				
0	<i>messages sent</i>				
0	<i>messages received</i>				
0	<i>signals received</i>				
233	<i>voluntary context switches</i>				
90	<i>involuntary context switches</i>				



- 1 The ARC tool
- 2 First manipulation : basics of ALTARICA
- 3 Second manipulation : validation with a model checker
- 4 Formal design of a lift
- 5 Controller synthesis of a tank

Informal description

A system of production is composed of :

- a cistern containing enough water to supply the operating.
- a tank,
- two upstream pipes connecting the cistern to the tank, for bringing water to the tank,
- one downstream pipe to drain the water from the tank,
- three controllable valves to increase or decrease the flow in each pipe,
- a software controller.

Requirements

- ① The level of the tank must remain in the intermediate zone.
- ② Depending on the number of down valves, and after a possible initial period as short as possible, the flow of the downstream valve should be as big as possible.

A valve

Technical description

- Three flow levels : 0, 1 et 2.
- Reacts to two signals : `inc` et `dec`
- Rust over time can block non perfect valve. The flow is no longer updatable.



A valve

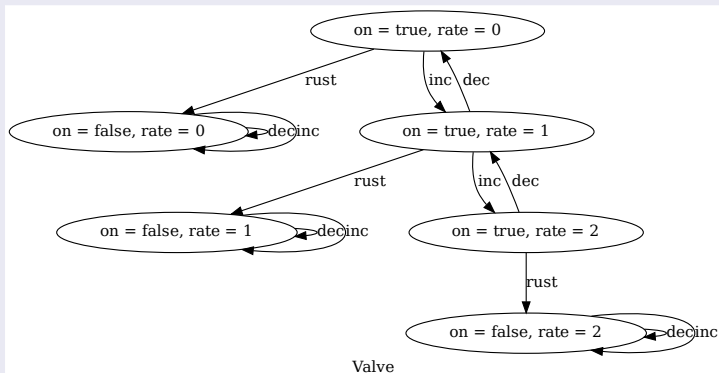
An ALTARICA model of a valve

```
node Valve
state
  rate : Rates : public;
  on : bool : private;
event
  inc, dec, rust;
trans
  on | - inc -> rate := rate+1;
  on | - dec -> rate := rate-1;
  on | - rust -> on := false;
  ~on | - inc, dec -> ;
init
  on := true, rate := LowRate;
edon
```



A valve

The ALTARICA semantic of the valve



A valve

Validation of the Valve

```
/*  
 * Properties for node : Valve  
 * # state properties : 2  
 *  
 * any_s = 6  
 * initial = 1  
 *  
 * # trans properties : 4  
 *  
 * any_t = 19  
 * self = 12  
 * epsilon = 6  
 * self_epsilon = 6  
 */  
TEST(deadlock,0)           [PASSED]  
TEST(notResettable,0)      [FAILED] actual size = 3
```

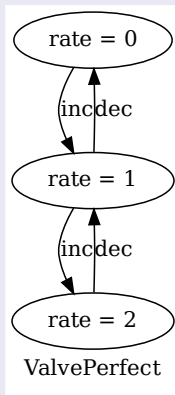


An ALTARICA model of a perfect valve

```
node ValvePerfect
  state
    rate : Rates : public;
  event
    inc, dec, rust;
  trans
    true  |- inc → rate := rate+1;
    true  |- dec → rate := rate-1;
    false |- rust → ;
  init
    rate := LowRate;
edon
```

A valve

The ALTARICA semantic of the perfect valve



A valve

Validation of the perfect Valve

```
/*  
 * Properties for node : ValvePerfect  
 * # state properties : 2  
 *  
 * any_s = 3  
 * initial = 1  
 *  
 * # trans properties : 4  
 *  
 * any_t = 7  
 * self = 3  
 * epsilon = 3  
 * self_epsilon = 3  
 */  
TEST(deadlock,0)           [PASSED]  
TEST(notResetable,0)       [PASSED]
```



The tank

The technical description

- Four sensors to define 5 zones numbered from 0 to 4.
- A maximal input rate of 4.
- A maximal output rate of 2.
- Physical rules to determine level evolution.
 - if $input > output$ then, in the future, the level will increase by 1.
 - if $input < output$ then, in the future, the level will decrease by 1.
 - if $input = output = 0$ then, in the future, the level will remain the same.
 - if $input = output > 0$ then the future is not predictable. The level may increase by 1, decrease by 1 or remain the same.



The tank


An ALTARICA model of tank

```
node Tank
  flow
    input : [LowRate+LowRate, HighRate+HighRate];
    output : Rates;
  state
    level : Levels : public;
  event
    time;
  trans
    (input=output) |- time -> ;
    (input=output) & (output>0) |- time -> level := level+1;
    (input=output) & (output>0) |- time -> level := level-1;
    (input>output) |- time -> level := level+1;
    (input>output) & (level=HighLevel) |- time -> ;
    (input<output) |- time -> level := level-1;
    (input<output) & (level=LowLevel) |- time -> ;
  init
    level := InitLevel;
edon
```



The tank

The ALTARICA semantic of the tank.

input in $[0, 4]$, output in $[0, 2]$, level in $[0, 6]$ 

Tank/Q (1 classes)

The tank

Validation of the tank

```
/*  
 * Properties for node : Tank  
 * # state properties : 2  
 *  
 * any_s = 105  
 * initial = 15  
 *  
 * # trans properties : 4  
 *  
 * any_t = 3510  
 * self = 138  
 * epsilon = 1575  
 * self_epsilon = 105  
 */  
TEST(deadlock,0)           [PASSED]  
TEST(notResetable,0)       [PASSED]
```



A very permissive controller

The technical description

- The controller can command the three valves at the same time.
- The controller observes the level into the tank.
- The controller observes all valve's rates.
- The controller don't observe valve's state.



A very permissive controller

An ALTARICA model of a controller

```
node ControllerPermissive
  // Les observations
  flow rate : Rates[3];
          level : Levels;
  // commands (one per valve) vector (Vup[0],Vup[0],Vdown)
  // d for dec, i for inc, n for nop
  event
    ddd, ddi, ddn, did, dii, din, dnd, dni, dnn,
    idd, idi, idn, iid, iii, iin, ind, ini, inn,
    ndd, ndi, ndn, nid, nii, nin, nnd, nni, nnn;
  trans
    true |—
      ddd, ddi, ddn, did, dii, din, dnd, dni, dnn,
      idd, idi, idn, iid, iii, iin, ind, ini, inn,
      ndd, ndi, ndn, nid, nii, nin, nnd, nni, nnn
    -> ;
edon
```



A very permissive controller

The ALTARICA semantic of the controller

level in [0, 6], rate[0] in [0, 2], rate[1] in [0, 2], rate[2] in [0, 2]

... ..

ControllerPermissiveQ (1 classes)

A very permissive controller

Validation of the controller

```
/*  
 * Properties for node : ControllerPermissive  
 * # state properties : 2  
 *  
 * any_s = 189  
 * initial = 189  
 *  
 * # trans properties : 4  
 *  
 * any_t = 1.00019e+06  
 * self = 5292  
 * epsilon = 35721  
 * self.epsilon = 189  
 */  
TEST(deadlock,0)           [PASSED]  
TEST(notResetable,0)       [PASSED]
```



The uncontrolled system

The technical description

Fairly standard assumptions for reactive systems :

- Controls do not take time.
- Between two failures and/or temporal evolutions, the controller has always time to give at least an order.
- Between two orders, the system has always time to react.



The ALTARICA code of the uncontrolled system I

```
node System
  sub
    Vup : Valve[2];
    Vdown : Valve;
    T : Tank;
    C : ControllerPermissive;
  assert
    // links between valves and the tank.
    T.input = (Vup[0].rate + Vup[1].rate);
    T.output = Vdown.rate;
    // Controller's observations of the system
    C.rate[0] = Vup[0].rate;
    C.rate[1] = Vup[1].rate;
    C.rate[2] = Vdown.rate;
    C.level = T.level;
  state
    ctrl : bool;
  init
    ctrl := true;
  event // Controller commands and system reactions
    command, reaction;
  trans
    ctrl |— command → ctrl := false;
```



The ALTARICA code of the uncontrolled system II

```
~ctrl |— reaction → ctrl := true;
sync  // The command effects.
<command, C.ddd, Vup[0].dec, Vup[1].dec, Vdown.dec>;
<command, C.ddi, Vup[0].dec, Vup[1].dec, Vdown.inc>;
<command, C.ddn, Vup[0].dec, Vup[1].dec>;
<command, C.did, Vup[0].dec, Vup[1].inc, Vdown.dec>;
<command, C.dii, Vup[0].dec, Vup[1].inc, Vdown.inc>;
<command, C.din, Vup[0].dec, Vup[1].inc>;
<command, C.dnd, Vup[0].dec, Vdown.dec>;
<command, C.dni, Vup[0].dec, Vdown.inc>;
<command, C.dnn, Vup[0].dec>;
<command, C.idd, Vup[0].inc, Vup[1].dec, Vdown.dec>;
<command, C.idi, Vup[0].inc, Vup[1].dec, Vdown.inc>;
<command, C.idn, Vup[0].inc, Vup[1].dec>;
<command, C.iid, Vup[0].inc, Vup[1].inc, Vdown.dec>;
<command, C.iii, Vup[0].inc, Vup[1].inc, Vdown.inc>;
<command, C.iin, Vup[0].inc, Vup[1].inc>;
<command, C.ind, Vup[0].inc, Vdown.dec>;
<command, C.ini, Vup[0].inc, Vdown.inc>;
<command, C.inn, Vup[0].inc>;
<command, C.ndd, Vup[1].dec, Vdown.dec>;
<command, C.ndi, Vup[1].dec, Vdown.inc>;
<command, C.ndn, Vup[1].dec>;
```



The ALTARICA code of the uncontrolled system III

```
<command, C.nid, Vup[1].inc, Vdown.dec>;
<command, C.nii, Vup[1].inc, Vdown.inc>;
<command, C.nin, Vup[1].inc>;
<command, C.nnd, Vdown.dec>;
<command, C.nni, Vdown.inc>;
<command, C.nnn>;
// An evolution without failure.
<reaction, T.time>;
// A (only one) failure may occurs during an evolution of
  the system.
<reaction, T.time, Vup[0].rust>;
<reaction, T.time, Vup[1].rust>;
<reaction, T.time, Vdown.rust>;
```

edon



Validation of the uncontrolled system

Validation results

```
/*  
 * Properties for node : System  
 * # state properties : 2  
 *  
 * any_s = 2725  
 * initial = 1  
 *  
 * # trans properties : 4  
 *  
 * any_t = 32344  
 * self = 2725  
 * epsilon = 2725  
 * self_epsilon = 2725  
 */  
TEST(deadlock,0)           [PASSED]  
TEST(notResetable,0)      [FAILED] actual size = 2370
```



Validation of the uncontrolled system

Specific validation results

```
/*  
 * Properties for node : System  
 * # state properties : 2  
 *  
 * lowLevel = 224  
 * highLevel = 393  
 *  
 * # trans property : 0  
 *  
 */
```



The ALTARICA code of the uncontrolled perfect system

Diff with the unperfect one

```
d1 1
a1 1
node SystemPerfect
d3 2
a4 2
    Vup : ValvePerfect[2];
    Vdown : ValvePerfect;
```



Validation of the uncontrolled perfect system

Validation results

```
/*  
  * Properties for node : SystemPerfect  
  * # state properties : 2  
  *  
  * any_s = 355  
  * initial = 1  
  *  
  * # trans properties : 4  
  *  
  * any_t = 2731  
  * self = 355  
  * epsilon = 355  
  * self_epsilon = 355  
*/  
TEST(deadlock,0)           [PASSED]  
TEST(notResetable,0)       [PASSED]
```



Validation of the uncontrolled perfect system

Specific validation results

```
/*  
 * Properties for node : SystemPerfect  
 * # state properties : 2  
 *  
 * lowLevel = 36  
 * highLevel = 50  
 *  
 * # trans property : 0  
 *  
 */
```



Specification for the controller's synthesis

The arena definition

```
with System, SystemPerfect, SystemControlled,  
    SystemPerfectControlled, SystemPerfectControlledOpt,  
    SystemMemory, SystemMemoryControlled do  
// Critical configurations  
ER := any-s & ([T.level=LowLevel | T.level=HighLevel] |  
    deadlock);  
// les types d'evenements  
control      := any-t & label command;  
uncontrol    := any-t & label reaction;  
// Initial loosing and winning positions for controller  
LossCtrl := src(control) & ER;  
WinCtrl  := src(control) - ER;  
// Initial loosing and winning positions for environment  
LossEnv  := src(uncontrol) - ER;  
WinEnv   := src(uncontrol) & ER;  
done
```



Specification for the controller's synthesis

The fix point definition

```
with System, SystemPerfect, SystemControlled,  
    SystemPerfectControlled, SystemPerfectControlledOpt,  
    SystemMemory, SystemMemoryControlled do  
WinningCtrl -=  
    control &  
    rtgt(LossEnv &  
        (src(uncontrol & rtgt(WinCtrl & src(WinningCtrl))) -  
         src(uncontrol - rtgt(WinCtrl & src(WinningCtrl)))));  
  
WinningEnv +=  
    uncontrol &  
    rtgt(LossCtrl |  
        (src(uncontrol & rtgt(WinEnv | src(WinningEnv))) -  
         src(uncontrol - rtgt(WinEnv | src(WinningEnv)))));  
done
```



Specification for the controller's synthesis

Results output

```
with System, SystemPerfect, SystemControlled,
    SystemPerfectControlled, SystemPerfectControlledOpt,
    SystemMemory, SystemMemoryControlled do
  // Is the system controllable ?
  uncontrollable := initial - src(WinningCtrl);
  // Is the system controlled ?
  uncontrolled := control - WinningCtrl;
  // Widened Controller generation
  project(any_s, (WinningCtrl|uncontrol,empty_t),
    'WController.$NODENAME', true, C)
    > 'Alt/WController.$NODENAME.alt';
  // Narrowed Controller generation
  project(any_s, (uncontrol,WinningCtrl),
    'NController.$NODENAME', true, C)
    > 'Alt/NController.$NODENAME.alt';
  // Widened Bug generation
  project(any_s, (WinningEnv|control,empty_t), 'WBug.$NODENAME',
    true)
    > 'Alt/WBug.$NODENAME.alt';
  // Narrowed Bug generation
  project(any_s, (control,WinningEnv), 'NBug.$NODENAME', true)
```



Results of controller's synthesis

Properties for the imperfect system

```
TEST(ER,0)          [FAILED] actual size = 617  
TEST(uncontrollable,0) [PASSED]  
TEST(uncontrolled,0)  [FAILED] actual size = 24201
```

Properties for the perfect system

```
TEST(ER,0)          [FAILED] actual size = 86  
TEST(uncontrollable,0) [PASSED]  
TEST(uncontrolled,0)  [FAILED] actual size = 894
```



Replacement of controllers

The controlled imperfect system

```
d1 1
a1 1
node SystemControlled
d6 1
a6 1
    C : Controller_System;
```

The controlled perfect system

```
d1 1
a1 1
node SystemPerfectControlled
d6 1
a6 1
    C : Controller_SystemPerfect;
```



Replacement of controllers

Properties for the controlled imperfect system

```
TEST(ER,0)          [FAILED] actual size = 72  
TEST(uncontrollable,0) [PASSED]  
TEST(uncontrolled,0)  [FAILED] actual size = 525
```

Properties for the controlled perfect system

```
TEST(ER,0)          [PASSED]  
TEST(uncontrollable,0) [PASSED]  
TEST(uncontrolled,0)  [PASSED]
```



Optimization of controllers

The optimized controlled perfect system

```
a0 3
/* This node is a variant of the result of a projection.
 * We introduce priorities between controlled events.
 */
d5 1
a5 1
node ControllerOpt_SystemPerfect
d37 1
a37 4
    // Vdown(d) < Vdown(n) < Vdown(i)
    {ddd, did, dnd, idd, iid, ind, ndd, nid, nnd} < {ddn, din,
        dnn, idn, iin, inn, ndn, nin, nnn};
    {ddn, din, dnn, idn, iin, inn, ndn, nin, nnn} < {ddi, dii,
        dni, idi, iii, ini, ndi, nii, nni};
trans
d65 28
```



Optimization of controllers

Properties for the optimized controlled perfect system

```
TEST(ER, 0)           [PASSED]  
TEST(uncontrollable, 0) [PASSED]  
TEST(uncontrolled, 0)  [PASSED]
```



The process

- We have to precise the rust failure behaviors.
- We have use non determinism to design the level evolution.
- We have to describe a first controller as a very permissive one.
- We have to write greatest fix point operators.
- We have to check is the computed controllers are enough or not (depends on the observable configurations).
- We use priorities to optimize the controller to satisfy the second requirement.



The result

- It is possible to control the level into the tank if valves never fail.
- It is possible to optimize the control in order to have the downstream valve always at its maximal rate.



Conclusion

Performances

NetBSD amd64 x86_64

7,32	<i>real</i>	3,85	<i>user</i>	3,34	<i>sys</i>
0	<i>maximum resident set size</i>				
0	<i>average shared memory size</i>				
0	<i>average unshared data size</i>				
0	<i>average unshared stack size</i>				
52540	<i>page reclaims</i>				
0	<i>page faults</i>				
0	<i>swaps</i>				
0	<i>block input operations</i>				
93	<i>block output operations</i>				
0	<i>messages sent</i>				
0	<i>messages received</i>				
0	<i>signals received</i>				
97	<i>voluntary context switches</i>				
119	<i>involuntary context switches</i>				

