Master Informatique	ANNÉE [YEAR] : 2007/2008 **Specialité [Specialty]:** Master 2 Informatique Génie Logiciel **UV :** V-INF 922 : Conception Formelle [Formal Design] **Date :** 2008 **Heure :** **Durée [Duration] :** 2 heures [2 hours] Documents : autorisés [authorized] Épreuve [Exam] de M. Alain GRIFFAULT

Name : Firstname :

Warnings

- Most of the queries are independent.
- Indication reflects (approximately) the difficulty of a question.
- The space allowed for the answers is enough (unless you use it as a draft, which is strongly discouraged).

Subject : Security by redundancy

The aim is the modeling and the specification of redundancy architectures, in order to verify that these techniques generate no gain and no loss of features.
The approach is the following:

1. Let a perfect server with a number of features.
2. We add dysfunctions (failure and reparation) in the server.
3. We build a system based on redundancy to mask some possible failures, and to have a better availability.
4. We specify the no gain and the no loss of features.
5. We check theses properties for different systems.

Requirements

Requirements specify that the redundancy don't change features.

P1 : Without failure, the system behaves like the perfect server.

P2 : With less than two failures at the same time, the system behaves like the perfect server.

P3 : With the possibility of two failures at the same time, the system behaves like the perfect server, but with some loss of availability.

1 The perfect server

1.1 The Server node

1.1.1 The AltaRica code

```
node Server
  state mode : [0,3] : parent;
  event a, b, c
  trans
    // features
    mode = 0 |- a -> mode := 1;
    mode = 1 |- b -> mode := 2;
    mode = 2 |- a -> mode := 1;
    mode = 2 |- c -> mode := 3;
    mode = 3 |- b -> mode := 0;
    mode = 3 |- b -> mode := 1;
  init
    mode := 0;
edon
```

1.1.2 The semantics

Question 1 (1 point) *Draw the graph of the semantics.*

1.2 The reality

Unfortunately, servers are not perfect in a real world.

2 The ServerV1 node

This server has the same features as the perfect server, but it can fail.
The repair can be done without modifying the system, and as soon as the server is repaired, its new mode is assigned and it restarts.

2.1 The AltaRica code

```
node ServerV1
  state mode : [0,3] : parent;
        ok   : bool  : parent;    // For the faliure
  flow  modeForRestart : [0,3];
  event a, b, c, failure, repairAndRestart
  trans
    // features are only possible if all is ok
    ok & mode = 0 |- a -> mode := 1;
    ok & mode = 1 |- b -> mode := 2;
    ok & mode = 2 |- a -> mode := 1;
    ok & mode = 2 |- c -> mode := 3;
    ok & mode = 3 |- b -> mode := 0;
    ok & mode = 3 |- b -> mode := 1;
    // failure can appear only when the server is in use.
    ok            |- failure -> ok := false;
    // the reparation restart the server from the mode given by the parameter.
    ~ok           |- repairAndRestart -> ok := true, mode := modeForRestart;
  init
    mode := 0, ok := true;
edon
```

2.2 The semantic

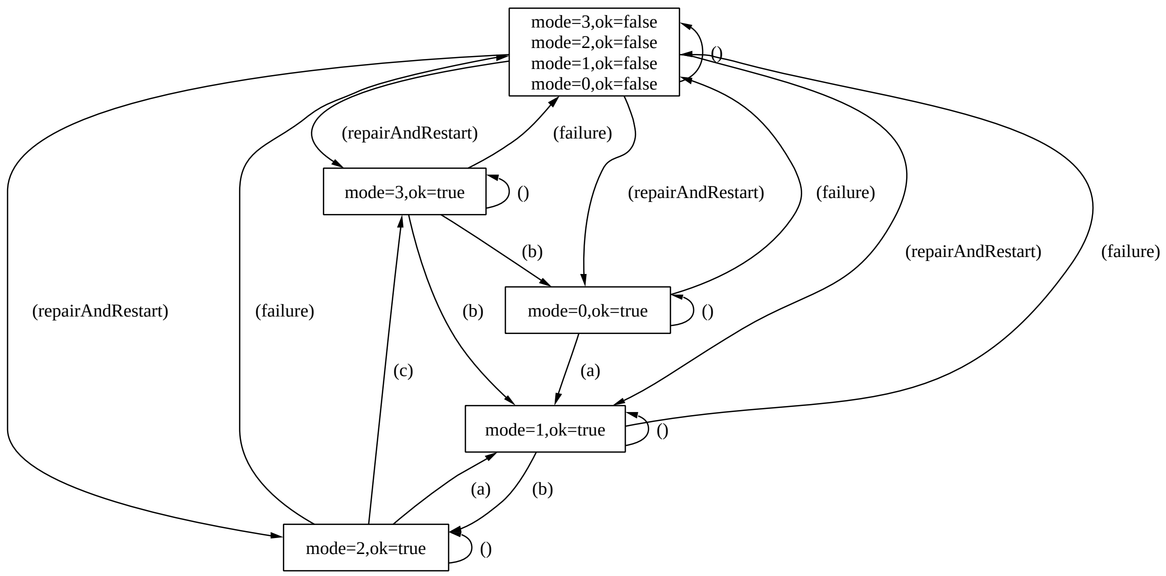

Figure 1: The ServerV1 node without the variable `modeForRestart`

3 The ServerV2 node

This server has the same features as the perfect server, but it can fail.
The repair can be done without modifying the system, and as soon as the server is repaired, it is ready to restart.
When the server is required, its new mode is assigned and it restarts.

3.1 The semantics

Question 2 (1 point) *Draw the expected graph for the semantics, without the variable* `modeForRestart`

3.2 The AltaRica code

Question 3 (3 points) *Complete the* AltaRica *code for your server.*

```
node ServerV2
  state mode : [0,3] : parent;




  flow  modeForRestart : [0,3];
  event a, b, c, failure, repair, start
  trans
    // functionalities are impossible if the server is breakdown
    mode = 0                                   |- a
      -> mode := 1                                                  ;
    mode = 1                                   |- b
      -> mode := 2                                                  ;
    mode = 2                                   |- a
      -> mode := 1                                                  ;
    mode = 2                                   |- c
      -> mode := 3                                                  ;
    mode = 3                                   |- b
      -> mode := 0                                                  ;
    mode = 3                                   |- b
      -> mode := 1                                                  ;

    // failure is only possible when the server is in use
                                               |- failure
       ->                                                           ;

    // the reparation
                                               |- repair
       ->                                                           ;

    // the server restarts in a mode given
    // by the modeForRestart "parameter"
                                               |- start
       ->                                                           ;

  init
    mode := 0
edon
```

4 The hot redundancy

The so-called hot redundancy is a technique that uses simultaneously two machines of type `ServerV1`, so that if one fails, the other is immediately operational.
In this technique, as the two servers are not down, they work together. When one of them fails, the other continues alone. Finally, as soon as (at the same time) a server is repaired, its mode is set to the correct value in order to restart immediately (at the same time).
One difficulty with this architecture appears when the two servers are down at the same time. The first server which is repaired must restart from the mode in which was the last server which breaks. This in order to simulate the perfect server.

4.1 The AltaRica code

Question 4 (3,5 points) *Complete this AltaRica code that describes two servers with hot redundancy.*

```
node SystemV1
  sub   S1, S2 : ServerV1
  state lastToShutdown : [1,2];
  flow  currentMode : [0,3];
  event failureS1, failureS2;
  trans
    // Memorization of the last server which breaks



  sync
    // features




    // failures
    <failureS1, S1.failure>;
    <failureS2, S2.failure>;
  assert
    // systems in use are in the same mode
    (S1.mode = S2.mode) | ~S1.ok | ~S2.ok;
    // Definition of the current mode






    // Definition of the two modeForRestart "parameters"


edon
```

5 The cold redundancy

The so-called cold redundancy is a technique that uses simultaneously two machines of type **ServerV2**, so that if one fails, the other is immediately operational.
In this technique, a maximum of one server works at the same time. When the running server fails, it shall communicate its current mode, and the second server must start in this mode. Finally, as soon as a server is repaired, it stands ready to start.
One difficulty with this architecture appears when the two servers are down at the same time. The first server which is repaired must restart from the mode in which was the last server which breaks. This in order to simulate the perfect server.

5.1 The AltaRica code

Question 5 (3,5 points) *Complete this* AltaRica *code that describes two servers with cold redundancy.*

```
node SystemV2
  sub   S1, S2 : ServerV2
  state lastToShutdown : [1,2];
  flow  currentMode : [0,3];
  event failureS1, failureS2;
  trans
    // Memorization of the last server which breaks



  sync
    // independant starts of servers



    // a failure restarts the other server (if it is possible)



  assert
    // Never two servers "on" at the same time.



    // Definition of the current mode









    // Definition of the two modeForRestart "parameters"


edon
```

6 Validity of redundancy

We want to show that the redundancy techniques are correct in the sense that they do not alter features. The redundant servers behave exactly like a perfect server if we don't observe failures and repairs.
We will use the `acheck` model-checker to do that.

6.1 Requirements

6.1.1 Specification

Question 6 (7 points) *Complete this* `acheck` *code to verify the hot redundancy.*

```
with SystemV1 do
 // Validation of the model
 quot()     >  '$NODENAME.dot';
 dead       := any_s - src(any_t - self_epsilon);
 notCFC     := any_t - loop(any_t,any_t);
 // A partition of the graph depending on the number of failures
 nbFailure0 :=
 nbFailure2 :=
 nbFailure1 :=
 // features
 a          := label S1.a | label S2.a;
 b          := label S1.c | label S2.b;
 c          := label S1.c | label S2.c;
 // dysfunctionalities
 failure          := label S1.failure | label S2.failure;
 repairAndRestart := label S1.repairAndRestart | label S2.repairAndRestart;
 /* P1 : Without failure, the system is correct (servers are in the same mode
  * WARNING : WE COMPUTE COUNTER EXAMPLES
  */
 ce_P1          :=

 /* P2 : With less than two failures, the system behaves perfectly
  * The property P2 is split with respect to features
  * P2 = P2a & P2b & P2c
  * P2a : With less than two failures, "a" feature behaves perfectly
  * P2a  is split in two parts
  * P2a = P2aFailure & P2aRepair
  * P2aFailure : failure event respects "a" feature
  *  1/ may "a" before "failure" => may "a" after "failure"
  *  2/ can't "a" before "failure" => can't "a" after "failure"
  * P2aRepair :  the repair event respects the "a" feature
  *  1/ may "a" before "repair" => may "a" after "repair"
  *  2/ can't "a" before "repair" => can't "a" after "repair"
```

```
 *
 * WARNING : WE COMPUTE COUNTER EXAMPLES
 */
ce_P2aFailure1 := nbFailure0 &
                  src(a) &
                  src(failure & rtgt(any_s - src(a)));
/* QUESTION : Make a schema to explain the previous calculus
 *
 *
 *
 *
 *
 *
 *
 *
 */
ce_P2aFailure2 :=


ce_P2aFailure  :=
ce_P2aRepair1  :=


ce_P2aRepair2  :=


ce_P2aRepair   :=
ce_P2a         :=

/* P3 : With the possibility of two failures at the same time,
 *      the system behaves like the perfect server,
 *      but with some loss of availability.
 * The property P3 is split with respect to features
 * P3 = P3a & P3b & P3c
 * P3a : Even with two failures, "a" feature is not gain and not loss
 * P3a is split in two parts
 *  1/ may "a" before the second "failure" => may "a" after "repair"
 *  2/ can't "a" before "repair" => can't "a" after the second "failure"
 *
 * WARNING : WE COMPUTE COUNTER EXAMPLES
 */
```

```
 ce_P3a1         := nbFailure1 &
                    src(a) &
                    src(failure & rtgt(src(repairAndRestart &
                                               rtgt(any_s - src(a)))));
 /* QUESTION : Make a schema to explain the previous calculus
  *
  *
  *
  *
  *
  *
  *
  *
  */
 ce_P3a2         :=



 ce_P3a          :=
 // Output of the results
 test(dead,0)   >  '$NODENAME.res';
 test(notCFC,0) >> '$NODENAME.res';
 test(ce_P1,0)     >> '$NODENAME.res';
 test(ce_P2a,0)    >> '$NODENAME.res';
 test(ce_P3a,0)    >> '$NODENAME.res';
done
```

Question 7 (1 point) *Complete this table. When your answer is* `[FAILED]`, *indicate whether it is due to a bad model or to a poor writing of the property.*

Requirement	[PASSED]	[FAILED]	Explanation
`TEST(dead=0)`			
`TEST(notCFC=0)`			
`TEST(P1=0)`			
`TEST(P2a=0)`			
`TEST(P3a=0)`			