

Sviluppo di Software Sicuro - S³ Condizioni di verifica - IV

Corso di Laurea Magistrale in
Sicurezza Informatica: Infrastrutture e Applicazioni
Università di Pisa – Polo di La Spezia
C. Montangero
Anno accademico 2009/10

Sommario

- Data refinement – introduzione
- Caratterizzazione del tipo astratto
- Definizione della rappresentazione
- Relazione tra tipo astratto e concreto
- Codifica del tipo concreto
- Verifica del codice

S³ 2009/10 – Condizioni di verifica – Parte III

INTRODUZIONE

S3: VC- C.Montangero - Copyright 2010

3

Il problema

- Mettere a disposizione un TDA
 - visione astratta di un tipo di dato, semplice da utilizzare
 - esempi:
 - Integer invece di sequenze di bit
 - stack invece di array e puntatore
- e realizzarlo in modo efficiente (accesso, memoria)
- ma... **corretto!**

S3: VC - C.Montangero - Copyright 2010

4

Procedimento

- Caratterizzazione del tipo astratto
 - sintassi: interface in Java, package spec in Ada
 - costanti, costruttori, selettori e predicati
 - semantica: modello, assiomi
 - significato alle operazioni, in senso astratto
- Definizione della rappresentazione
 - struttura dati concreta
 - variabili private di una classe implements, private in Ada
- Relazione tra tipo astratto e concreto
 - corrispondenza matematica tra i due insiemi di valori
- Codifica del tipo concreto
 - classe in Java, package body in Ada
- Verifica del codice

S3: VC - C.Montangero - Copyright 2010

5

S³ 2009/10 – Condizioni di verifica – Parte III

CARATTERIZZARE UN TIPO ASTRATTO

S3: VC- C.Montangero - Copyright 2010

6

Esempio: IntStack

```
package Stack
  --# own State ;
  --# initializes State;
is
  function Empty return Boolean;
  --# global State;
  --# return ? ;

  procedure Clear;
  --# global out State;
  --# derives State from ;
  --# post ? ;

  procedure Pop (X : out Integer);
  --# global in out State;
  --# derives State from *
  --# &      X      from State;
  --# pre ? ;
  --# post ? ;
```

- Come diciamo:
 - il risultato di Empty è true se lo stack è vuoto?
 - lo stato dopo Clear è ure vuoto?
 - ^Pop richiede che lo stack non sia vuoto?
 - ^Pop assicura che X vale ciò che stava in cima allo stack, e che X è stato eliminato?

S3: SPARK - C.Montangero - Copyright 2010

7

Modello (canonico)

- Il tipo StackT (il tipo dei valori che può assumere State) è così definito:
 - "nil" è uno stack
 - se s è uno stack, e X un intero, "push(?X,?s)" è uno stack
 - non ci sono altri stack

```
nil
... push(-1,nil) push(0,nil) push(1,nil) ...
... push(- 1, push(-1,nil)) push(0,push(-1,nil)) ...
...
```

- Semantica delle costanti e dei costruttori.

S3: SPARK - C.Montangero - Copyright 2010

8

Modello (canonico) (2)

- Selettori e predicati vengono definiti sul modello
 - `empty(s) = true` se `s="nil"`, false altrimenti
 - `pop(s) = s'` se `s="push(?X,?s)"` per qualche `X`
 - `top(s) = X` se `s="push(?X,?s)"` per qualche `s'`
- Potremmo allora definire le procedure
(NB Ada nomi maiuscoli, modello minuscoli)

S3: SPARK - C.Montangero - Copyright 2010

9

Esempio: IntStack

```
package Stack
--# own State ;
--# initializes State;
is
  function Empty return Boolean;
  --# global State;
  --# return State = "nil" ;

  procedure Clear;
  --# global out State;
  --# derives State from ;
  --# post State = "nil" ;

  procedure Pop (X : out Integer);
  --# global in out State;
  --# derives State from *
  --# &      X      from State;
  --# pre  State /= "nil" ;
  --# post State = "push(?v,?s)"
  --#      -> X = v and State = s ;
```

- Come diciamo:
 - il risultato di `Empty` è true sse lo stack è vuoto?
 - lo stato dopo `Clear` è ure vuoto?
 - `^Pop` richiede che lo stack non sia vuoto?
 - `^Pop` assicura che `X` vale ciò che stava in cima allo stack, e che `X` è stato eliminato?

S3: SPARK - C.Montangero - Copyright 2010

10

Modello assiomatico

- Il tipo StackT è definito implicitamente
- Dalle proprietà delle operazioni
 - nil: \rightarrow StackT empty: StackT \rightarrow Boolean
 - push: Int x StackT \rightarrow StackT
 - pop : StackT \rightarrow StackT top : StackT \rightarrow Int
- come
 - empty(nil) = true empty(push(X,s)) = false
 - pop(push(X,s)) = s
 - top(push(X,s)) = X

S3: SPARK - C.Montangero - Copyright 2010

11

Esempio: IntStack

```

package Stack
--# own State ;
--# initializes State;
is
function Empty return Boolean;
--# global State;
--# return empty(State) ;

procedure Clear;
--# global out State;
--# derives State from ;
--# post empty(State) ;

procedure Pop (X : out Integer);
--# global in out State;
--# derives State from *
--# &      X      from State;
--# pre not empty(State);
--# post State~ = push(Y,State)
--#      -> X = Y;

```

- Come diciamo:
 - il risultato di Empty è true sse lo stack è vuoto?
 - lo stato dopo Clear è ure vuoto?
 - ^Pop richiede che lo stack non sia vuoto?
 - ^Pop assicura che X vale ciò che stava in cima allo stack, e che X è stato eliminato?

S3: SPARK - C.Montangero - Copyright 2010

12

Pregi e difetti

- Modello canonico è meno astratto
- Gli assiomi corrono il rischio di essere incompleti o contraddittori
- Sotto certe condizioni dagli assiomi si ricava un modello canonico (molto simile a quello visto)
- Sul modello canonico si possono verificare le proprietà assiomatiche (modello di una teoria logica)
- e SPARK ?

S3: VC - C.Montangero - Copyright 2010

13

Tipi astratti in SPARK

- Annotazioni per dichiarare
 - tipi astratti
 - funzioni (di prova, esplicite)
- Esempi

```
--# type StackAT is abstract;

--# function push_a(X : Integer; S : StackAT)
--#           return StackAT;
```
- Ma nessun modo di dare semantica...

Esempio IntStack

```

package Stack
  --# own State : StackAT;    -- promessa!
  --# initializes State;
is
  --# type StackAT is abstract;
  --# function empty_a(S: StackAT) return Boolean;
  --# function push_a(X : Integer; S : StackAT)
  --#           return StackAT;
  --# function pop_a(S : StackAT) return StackAT;
  --# function top_a(S : StackAT) return Integer;
  --# function size_a(S : StackAT) return Natural;

  MaxSize : constant Natural := 100;

```

S3: VC- C.Montangero - Copyright 2010

15

con queste definizioni

```

is
  function Empty return Boolean;
  --# global State;
  --# return empty_a(State);

  procedure Clear;
  --# global out State;
  --# derives State from *
  --# post empty_a(State) ;

  procedure Pop (X : out Integer);
  --# global in out State;
  --# derives State from *
  --# & X from State;
  --# pre not empty_a(State);
  --# post State~ = push_a(X, State);

```

- Come diciamo:
 - il risultato di Empty è true sse lo stack è vuoto?
 - lo stato dopo Clear è ure vuoto?
 - ^Pop richiede che lo stack non sia vuoto?
 - ^Pop assicura che X vale ciò che stava in cima allo stack, e che X è stato eliminato?

S3: SPARK - C.Montangero - Copyright 2010

16

S³ 2009/10 – Condizioni di verifica – Parte III

DEFINIZIONE DELLA RAPPRESENTAZIONE

S3: VC- C.Montangero - Copyright 2010

17

Finora...

```
package body Stack
--# own State is Pointer, Vector;
is
  subtype PointerT is Natural
    range Natural'First .. MaxSize;
  Pointer : PointerT;

  type IntVectorT is array (Integer range <>) of Integer;
  subtype StackRange is Positive range 1 .. MaxSize;
  subtype IntStackT is IntVectorT (StackRange);
  Vector : IntStackT;
  EmptyVector : constant IntStackT :=
    IntStackT'(StackRange => 0);
```

- Un valore concreto è una coppia di valori
- Efficiente, ma difficile da trattare

S3: VC- C.Montangero - Copyright 2010

18

Il tipo concreto

```
type IntStackCT is record
  point : PointerT;      -- come sopra
  vect  : IntStackT;     -- come sopra
end record;

ConcreteStack : IntStackCT;
EmptyConcreteStack : constant IntStackCT :=
  IntStackCT'(point => 0, vect => EmptyVector);
```

- e quindi

```
package body Stack
--# own State is ConcreteStack; -- rappresentazione
```

- un valore concreto per ogni astratto

S3: VC - C.Montangero - Copyright 2010

19

S³ 2009/10 – Condizioni di verifica – Parte III

RELAZIONE ASTRATTO-CONCRETO

S3: VC - C.Montangero - Copyright 2010

20

Idealmente

- Siano
 - A il tipo astratto, con valori a, a', \dots
 - C il tipo concreto, con valori c, c', \dots
 - $\alpha: C \rightarrow A$, la funzione di astrazione
 - $\gamma: A \rightarrow C$, quella di concretizzazione
- con le condizioni (ovvie)
 - $\alpha(\gamma(a)) = a \quad \gamma(\alpha(c)) = c$
- ogni $P^A: A \rightarrow \text{Bool}$ definisce $P^C: C \rightarrow \text{Bool}$
- $P^C = \alpha \circ P^A$ (viceversa $P^A = \gamma \circ P^C$)
- Sia \mathcal{T} questa trasformazione applicata a pre/post condizioni

S3: VC - C.Montangero - Copyright 2010

21

Integrità del raffinamento

```

procedure P(X : C)
  --# pre   ^PC;
  --# post  PC^;

```

- *raffina onestamente (integrity)*

```

procedure P(X : A)
  --# pre   ^PA;
  --# post  PA^;

```

- purchè

$$\mathcal{T} \ ^P^A \rightarrow \ ^P^C$$

$$\mathcal{T} \ P^C^ \rightarrow P^A^$$

S3: VC - C.Montangero - Copyright 2010

22

Correttezza della realizzazione

```
procedure P(X : C)
--# pre  ^PC;
--# post PC^;
```

- *implementa* correttamente

```
procedure P(X : A)
--# pre  ^PA;
--# post PA^;
```

- purchè *il raffinamento sia integro*, e P^C sia *corretta*, ossia valga, al solito:

$$^P^C \rightarrow wp[\text{body}]P^C^$$

S3: VC - C.Montangero - Copyright 2010

23

Praticamente in Spark...

- La correttezza del raffinamento si esprime
 - con *rule* in file .rlu
 - che Simplifier usa per collegare
 - le pre/post-condizioni della specifica e del body
- Si tratta di una *specifica contratta* di \mathcal{T} , α e γ
- A *completo carico* del progettista sw!
- Questo risolve anche la questione delle due variabili
 - si assume a livello astratto, in FDL

S3: VC - C.Montangero - Copyright 2010

24

Raffinamento per Examiner

- La dichiarazione (nel body)


```
--# own State is Pointer, Vector;
```
- viene interpretata da Examiner come un raffinamento: genera vc per collegare le pre/post condizioni di specifica e body, nella forma

$$\wedge p^A \rightarrow \wedge p^C \quad e \quad p^C \wedge \rightarrow p^A \wedge$$

- Le regole servono per semplificarle

S3: VC - C.Montangero - Copyright 2010

25

Integrità del raffinamento

```
procedure Pop(X: out Integer);  Pop (X : out Integer)
--# global in out State;          in out Pointer; in Vector;
...                               ...
--# pre    not empty_a(State);    Pointer > 0 ;
```

For checks of refinement integrity:

procedure_pop_4.

H1: not empty_a(state) .

->

C1: fld_pointer(state) > 0 .

S3: VC - C.Montangero - Copyright 2010

26

Integrità del raffinamento

```

procedure Pop(X: out Integer);  Pop (X : out Integer)
--# global in out State;          in out Pointer; in Vector;
...
--# pre    not empty_a(State);    Pointer > 0 ;

```

For checks of refinement integrity:

procedure_pop_4.

H1: not empty_a(state) .

->

C1: fld_pointer(state) > 0 .

fld sta per **field** :

fld_pointer è quindi il campo pointer del
record che rappresenta lo stack;

NB: state è implicitamente astratto in H1,
concreto in C1.

S3: VC - C.Montangero - Copyright 2010

27

Integrità del raffinamento

```

procedure Pop(X: out Integer);  Pop (X : out Integer)
--# global in out State;          in out Pointer; in Vector;
...
--# pre    not empty_a(State);    Pointer > 0 ;

```

For checks of refinement integrity:

procedure_pop_4.

H1: not empty_a(state) .

->

C1: fld_pointer(state) > 0 .

\mathcal{T} not empty_a(state) -> fld_pointer(state) > 0 ?

S3: VC - C.Montangero - Copyright 2010

28

Integrità del raffinamento

For checks of refinement integrity:

procedure_pop_4.

H1: not empty_a(state) .

->

C1: fld_pointer(state) > 0 .

Pop (X : out Integer)

in out Pointer; in Vector;

...

Pointer > 0 ;

\mathcal{T} not empty_a(state)

\equiv

not \mathcal{T} empty_a(state)

\equiv converto e applico la funzione concreta

not (\mathcal{T} empty_a)(y state)

$\equiv \lambda x. (\mathcal{T} \text{ empty_a}) = \lambda x. (\text{fld_pointer}(x) = 0)$, per definizione di raffinamento

not ($\text{fld_pointer}(y \text{ state}) = 0$)

In SPARK, si abbrevia il tutto con le regole utente, in un file .rlu:

empty_a(X) may_be_replaced_by fld_pointer(X) = 0

S3: VC - C.Montangero - Copyright 2010

29

Integrità del raffinamento

```

procedure Pop(X: out Integer);      Pop (X : out Integer)
--# global in out State;              in out Pointer; in Vector;
...                                  ...
--# post State=pop_a(State~) and    --# post Pointer=Pointer~ -1
--#      X = top_a(State~);          --# and X = Vector(Pointer~);

```

procedure_pop_5.

H1: not empty_a(state~) .

H3: pointer~ = fld_pointer(state~) .

H4: fld_vector(state~) = fld_vector(state) .

->

C1: state = pop_a(state~) .

C2: element(fld_vector(state~), [pointer~]) = top_a(state~) .

S3: VC - C.Montangero - Copyright 2010

30

Integrità del raffinamento

```

 $\mathcal{T}$  (= (X, Vector(Pointer~)))
≡  def: top_c =  $\lambda x$ .element(fld_vector(x) ,[fld_pointer(x)])
    e H4
 $\mathcal{T}$  (= (X, top_c(mk(Pointer~, Vector~))))
≡
( $\mathcal{T}$  =) (X, ( $\mathcal{T}$  top_c)  $\alpha$  mk(Pointer~, Vector~))
≡   $\lambda x$ .( $\mathcal{T}$  top_c) =  $\lambda x$ .top_a      (def raff)
= (X, top_a(State~))

```

element(fld_vector(X), [fld_pointer(X)]) may_be_replaced_by top_a(X)

```

procedure_pop_5.
H1:  not empty_a(state~) .
H3:  pointer~ = fld_pointer(state~) .
H4:  fld_vector(state~) = fld_vector(state) .
->
C2:  element(fld_vector(state~), [pointer~]) = top_a(state~) .

```

S3: VC - C.Montangero - Copyright 2010

31

S³ 2009/10 – Condizioni di verifica – Parte IV

SIMPLIFIER: REGOLE UTENTE

S3: VC- C.Montangero - Copyright 2010

32

file .rlu

- uno per il progetto (e.g. stack.rlu)
 - con le regole di raffinamento
- uno per ogni sottoprogramma (e.g. pop.rlu)
 - con eventuali regole specifiche
- utilizzati dopo le semplificazioni automatiche
 - regole mirate alle condizioni restanti in .siv
 - alcune tattiche possono essere inibite: utile?

S3: VC - C.Montangero - Copyright 2010

33

Regole d'inferenza

- rulename(#): Goal **may_be_deduced_from** Conditions .
- f_property(2): $f(X,Y) \geq Z$ may_be_deduced_from
- $[X \geq W, Y \geq W, W \geq Z]$.
- Goal e ogni Condition sono termini Prolog (espressi in FDL)
- Le variabili (maiuscole) sono dette wildcards (jolly)
 - assegnamento per pattern matching
- f_property(2) è utile per C1: $f(a, b * b) \geq c + 1$.
- ma non per C1: $f(a, b * b) > c + 1$.
- the Simplifier will attempt to find a means of instantiating all of the wildcards in Goal such that it becomes an exact match for an *existing undischarged conclusion*: the main operator of the goal formula in the rule must be the same as that of the conclusion if we are to be able to pattern-match the two together.
- poi si procede con le condizioni: per C1
 - $a \geq W, b * b \geq W, W \geq c+1$

S3: VC - C.Montangero - Copyright 2010

34

Regole di riscrittura

- rulename(#): Old may_be_replaced_by New if Conditions.
 - parte if opzionale
 - numeri ovviamente diversi all'interno della famiglia
- If Old can be pattern-matched to the toplevel conclusion formula, this is a match.
- Otherwise, we look at its immediate subexpressions, and try pattern-matching against these in turn (in left-to-right order), and to the subexpressions of these expressions, and so on.
- Finding a match will instantiate the wildcards in Old, and will generally instantiate some (or all) of those in New and in the Conditions list.
- La sostituzione è

S3: VC - C.Montangero - Copyright 2010

35

Integrità del raffinamento

```

 $\mathcal{T}$  (= (Pointer, Pointer~ -1))
 $\equiv$  H4 (Vector è in modo in), H3
 $\mathcal{T}$  =(mk(Pointer, Vector), mk(Pointer~ -1, Vector~))
 $\equiv$ 
( $\mathcal{T}$  =)( $\alpha$  mk(Pointer, Vector),  $\alpha$  mk(Pointer~ -1, Vector~))
 $\equiv$  pop_a(S) =  $\alpha$  mk(fld_pointer(S)-1, fld_vector(S)) (def)
= (State, pop_a(State~))

```

X = pop_a(XT) may_be_deduced_from [] ? -- già astratto?

```

procedure_pop_5.
H1: not empty_a(state~) .
H3: pointer~ = fld_pointer(state~) .
H4: fld_vector(state~) = fld_vector(state) .
->
C1: state = pop_a(state~) .
C2: element(fld_vector(state~), [pointer~]) = top_a(state~) .

```

S3: VC - C.Montangero - Copyright 2010

36

Esempio: door.ads

```
package Door
--# own State : StateType;
--#      in Input;
is
  type T is (Open, Closed);
  --# type StateType is Abstract;
  --#
  --# function prf_alarmTimeout(DoorState : StateType)
  --#      return Clock.TimeT;
  -- non c'è bisogno di esportarlo, a differenza di:
  function TheCurrentDoor return T;
  --# global State;
  function TheDoorAlarm return AlarmTypes.StatusT;
  --# global State;
```

S3: VC - C.Montangero - Copyright 2010

37

Esempio: lockdoor.ads

```
--# post
--# -- PROOF ANNOTATIONS FOR SECURITY PROPERTY 3
--# -- After each call, the security property holds:
--#
--# ( (TheCurrentDoor(State) = Open and
--#   Latch.IsLocked(Latch.State) and
--#   Clock.GreaterThanOrEqual(
--#     Clock.TheCurrentTime(Clock.CurrentTime),
--#     prf_alarmTimeout(State))
--#   )
--#   <-> TheDoorAlarm(State) = AlarmTypes.Alarming
--# ) and
--# Latch.IsLocked(Latch.State);
```

S3: VC - C.Montangero - Copyright 2010

38

Esempio: door.adb

```
package body Door
--# own State is CurrentDoor,
--#           AlarmTimeout,
--#           DoorAlarm &
--#           Input is in Door.Interface.Input;
is
  CurrentDoor   : T;
  DoorAlarm     : AlarmTypes.StatusT;
  AlarmTimeout  : Clock.TimeT;
```

S3: VC - C.Montangero - Copyright 2010

39

Esempio: lockdoor.adb

```
--# post
--# -- PROOF ANNOTATIONS FOR SECURITY PROPERTY 3
--# -- After each call, the security property holds:
--#
--# ( (CurrentDoor = Open and
--#   Latch.IsLocked(Latch.State) and
--#   Clock.GreaterThanOrEqual(
--#     Clock.TheCurrentTime(Clock.CurrentTime),
--#     AlarmTimeout)
--#   )
--#   <-> DoorAlarm = AlarmTypes.Alarming
--# ) and
--# Latch.IsLocked(Latch.State);
```

S3: VC - C.Montangero - Copyright 2010

40

Esempio: lockdoor.adb vs .ads

```
--# ( (CurrentDoor          = Open and
      TheCurrentDoor(State)
--#   Latch.IsLocked(Latch.State) and
--#   Clock.GreaterThanOrEqualTo(
--#       Clock.TheCurrentTime(Clock.CurrentTime),
--#       AlarmTimeout      )
--#       prf_alarmTimeout(State)
--#   )
--#   <-> DoorAlarm = AlarmTypes.Alarming
--#       TheDoorAlarm(State)
--# ) and
--# Latch.IsLocked(Latch.State);
```

S3: VC - C.Montangero - Copyright 2010

41

Esempio: lockdoor.adb vs .ads

```
--# ( (fld_currentdoor(State) = Open and
      TheCurrentDoor(State)
--#   Latch.IsLocked(Latch.State) and
--#   Clock.GreaterThanOrEqualTo(
--#       Clock.TheCurrentTime(Clock.CurrentTime),
--#       fld_alarmtimeout(State)
--#       prf_alarmTimeout(State)
--#   )
--#   <-> fld_dooralarm(State) = AlarmTypes.Alarming
--#       TheDoorAlarm(State)
--# ) and
--# Latch.IsLocked(Latch.State);
```

S3: VC - C.Montangero - Copyright 2010

42

Esempio: VC di raffinamento post di lockdoor

```

C1:  latch__islocked(latch__state) and
      thecurrentdoor(state) = open and
      clock__greaterthanorequal(
        clock__thecurrenttime(clock__currenttime),
        prf_alarmtimeout(state))
    <->
      thedooralarm(state) = alarmtypes__alarming .

H9:  fld_currentdoor(state~) = open and
      latch__islocked(latch__state) and
      clock__greaterthanorequal(
        clock__thecurrenttime(clock__currenttime),
        fld_alarmtimeout(state))
    <-> fld_dooralarm(state) = alarmtypes__alarming .

H1:  fld_currentdoor(state~) = fld_currentdoor(state) .

```

S3: VC - C.Montangero - Copyright 2010

43

Esempio: usando H1

```

C1:  latch__islocked(latch__state) and
      thecurrentdoor(state) = open and
      clock__greaterthanorequal(
        clock__thecurrenttime(clock__currenttime),
        prf_alarmtimeout(state))
    <->
      thedooralarm(state) = alarmtypes__alarming .

H9:  fld_currentdoor(state) = open and
      latch__islocked(latch__state) and
      clock__greaterthanorequal(
        clock__thecurrenttime(clock__currenttime),
        fld_alarmtimeout(state))
    <->
      fld_dooralarm(state) = alarmtypes__alarming .

```

S3: VC - C.Montangero - Copyright 2010

44

Esempio: è proposizionale

```

C1:   a and
      (thecurrentdoor(state) = open and
       clock__greaterthanorequal(
         clock__thecurrenttime(clock__currenttime),
         prf_alarmtimeout(state))
      )
      <-> thedooralarm(state) = alarmtypes__alarming .

H9:   fld_currentdoor(state) = open and
      (a and
       clock__greaterthanorequal(
         clock__thecurrenttime(clock__currenttime),
         fld_alarmtimeout(state))
      )
      <-> fld_dooralarm(state) = alarmtypes__alarming .

```

S3: VC - C.Montangero - Copyright 2010

45

Esempio: è proposizionale

```

C1:   a and
      (b and
       clock__greaterthanorequal(
         clock__thecurrenttime(clock__currenttime),
         prf_alarmtimeout(state))
      )
      <-> thedooralarm(state) = alarmtypes__alarming .

H9:   b and
      (a and
       clock__greaterthanorequal(
         clock__thecurrenttime(clock__currenttime),
         fld_alarmtimeout(state))
      )
      <-> fld_dooralarm(state) = alarmtypes__alarming .

```

S3: VC - C.Montangero - Copyright 2010

46

Esempio: è proposizionale

```

C1:  a and
      (b and
        c)
      )
      <-> thedooralarm(state) = alarmtypes__alarming .

H9:  b and
      (a and
        c)
      )
      <-> fld_dooralarm(state) = alarmtypes__alarming .

```

S3: VC - C.Montangero - Copyright 2010

47

Esempio: è proposizionale

```

C1:  a and b and c      H9:  b and a and c
      <-> d .             <-> d .

```

- per cui

```

lockdoor_user(3):
  B1 and
    (thecurrentdoor(S2) = open and
      clock__greaterthanorequal(T, prf_alarmtimeout(S2)))
  <->
    thedooralarm(S2) = alarmtypes__alarming
  may_be_deduced_from
  [ fld_currentdoor(S) = open and
    (B1 and
      clock__greaterthanorequal(T, fld_alarmtimeout(S2)))
    <->
      fld_dooralarm(S2) = alarmtypes__alarming ,

    fld_currentdoor(S) = fld_currentdoor(S2) ] .

```

S3: VC - C.Montangero - Copyright 2010

48

S³ 2009/10 – Condizioni di verifica – Parte IV

RAFFINAMENTO DI VARIABILI ESTERNE

S3: VC- C.Montangero - Copyright 2010

49

Attributi di prova impliciti

- Per ogni variabile esterna Vin in modo **in**
 - Vin'Tail definisce l'effetto dell'assegnamento della variabile esterna a una interna
- Per ogni variabile esterna Vout in modo **out**
 - Vin'Append definisce l'effetto dell'assegnamento alla variabile esterna di una interna

S3: VC- C.Montangero - Copyright 2010

50

Tipico sensore

```
package Sensor
--# own in Stream;
is
procedure Read(X : out Integer);
--# global Stream;
--# derives X from Stream;
end Sensor;
```

- Se nelle prove è coinvolto solo l'ultimo valore letto, basta

```
--# post X = Stream~;
```

S3: VC - C.Montangero - Copyright 2010

51

Controllo del valore massimo

```
Limit : constant Integer := 100;
procedure AboveLimit(Result : out Boolean)
--# global Sensor.Stream;
--# derives Result from Sensor.Stream;
--# post Result <-> (Sensor.Stream~ > Limit);
is
  Reading : Integer;
begin
  Sensor.Read(Reading);
  Result := Reading > Limit;
end AboveLimit;
```

- con le vc seguenti

S3: VC - C.Montangero - Copyright 2010

52

For path(s) from start to finish:

```
procedure_abovelimit_1.
H1: true .
H2: sensor__stream >= integer__first .
H3: sensor__stream <= integer__last .
H4: reading__1 = sensor__stream .
->
C1: (reading__1 > 100) = (sensor__stream > 100) .
```

- che viene semplificata

S3: VC - C.Montangero - Copyright 2010

53

Valori crescenti?

```
procedure Rising(Result : out Boolean)
--# global Sensor.Stream;
--# derives Result from Sensor.Stream;
--# post Result <->
--#   Sensor.Stream'Tail(Sensor.Stream~)>Sensor.Stream~;
is
  Reading1, Reading2 : Integer;
begin
  Sensor.Read(Reading1);
  Wait.Ten;
  Sensor.Read(Reading2);
  Result := Reading2 > Reading1;
end Rising;
```

- con le vc

S3: VC - C.Montangero - Copyright 2010

54

Valori crescenti?

```
procedure_rising_1.  
H1: true .  
H2: sensor__stream >= integer__first .  
H3: sensor__stream <= integer__last .  
H4: reading1__1 = sensor__stream .  
H5: sensor__stream__1 = sensor__nextreading(sensor__stream) .  
H6: reading2__2 = sensor__stream__1 .  
H7: sensor__stream__2 = sensor__nextreading(sensor__stream__1).  
->  
C1: (reading2__2 > reading1__1) <->  
    (sensor__nextreading(sensor__stream) > sensor__stream) .
```

- con le vc

S3: VC - C.Montangero - Copyright 2010

55

- il nome X^{\sim} diventa 'x~' in Prolog (file .rlu)

S3: VC - C.Montangero - Copyright 2010

56