Falluto2.0.

UN MODEL CHECKER PARA SISTEMAS TOLERANTES A FALLAS

Raúl. Monti Pedro R. D'Argenio

Facultad de Matemática, Astronomía y Física, Universidad Nacional de Córdoba

Córdoba, Argentina. 2013

Tabla de contenidos

- **Sistemas críticos.**

  Ejemplos: aviónica, equipos médicos, etc...

- **Fallas (Fault) != Errores (Failure)**

  Falla = cambio de estado que "desestabiliza" al sistema.

  Error = desviación del comportamiento esperado.

- **Sistemas tolerantes a fallas.**

  No podemos evitar las fallas $\longrightarrow$ las superamos.

Model Checking.

- Método formal para la verificación de propiedades sobre sistemas.
- Exhaustivo sobre el espacio de estados del modelo finito.
- NuSMV es un model checker basado en BDD.

Motivación

```
int y1 = 0;
int y2 = 0;
short in_critical = 0;
```

```
active proctype process_1() {
    do
    :: true ->
        y1 = y2+1;
        ((y2==0) || (y1<=y2));
        in_critical++;
        in_critical--;
        y1 = 0;
    od
}
```

```
active proctype process_2() {
    do
    :: true ->
        y2 = y1+1;
        ((y1==0) || (y2<y1));
        in_critical++;
        in_critical--;
        y2 = 0;
    od
}
```

Objetivos

- Continuar con el trabajo hecho en Falluto(Hames) y Offbeat(Bordenabe).
- Definir un lenguaje práctico para el modelado y verificación de sistemas tolerantes a fallas, que oculte las funcionalidad de las fallas y provea facilidades para la especificación de propiedades.
- Elaborar un front-end para **NuSMV** que use este lenguaje.

Estructuras de Kripke

Una estructura de Kripke sobre AP se define como una 4-upla M = (S, I, R, L) donde

- S es un conjunto finito de estados
- $I \subseteq S$ estados iniciales
- $R \subseteq S \times S$ relación de transición left-total.
- L: $S \to P(AP)$ función de etiquetado o interpretación.

En model checking usualmente sucede que:

- S es un conjunto de valuaciones sobre las variables del sistema.
- AP es un conjunto de expresiones booleanas sobre las variables.
- $L(v) = \{a \in AP \mid v(a) \text{ es verdadero}\}$ con $v \in S$
- R explica la relación entre la valuación actual y la próxima

Ejemplo de estructura de Kripke

LTS

Un LTS es una 4-upla $M = (S, S_0, L, R)$ donde:

- S es un conjunto de estados
- $S_0 \subseteq S$ es un conjunto de estados iniciales
- L es un conjunto de etiquetas (nombres de transiciones)
- $R \subseteq S \times L \times S$ es una relación ternaria de transiciones etiquetadas

Si $s1, s2 \in S$, $l \in L$, y $(s1, l, s2) \in R$, entonces existe una transición con nombre l desde el estado $s1$ al estado $s2$.

Ejemplo de LTS

El lenguaje de NuSMV

Modelado

```
MODULE main()
  VAR
    v1:boolean;
    v2:{1,2,a,b};

  INIT
    v1 & v2 = a

  TRANS
    ( v1 -> next(v2) in {a,b}
    & !v1 -> next(v2) in {1,2}
    )
    | next(v1) = !v1
```

Propiedades

```
LTLSPEC G v1 = FALSE

CTLSPEC AG TRUE
```

Fairness

```
FAIRNESS v2 in {1,a}

COMPASSION (v2 = 1 , !v1)
```

El lenguaje de Falluto2.0

Modelado

```
PROCTYPE proc( cxtv1, ctxv2
             ; sync1, sync2)
  VAR
    var1 : bool

  INIT
    {formula}

  TRANS
    [name]: enable => changes
    [sync1] ...
    []: ...
ENDPROCTYPE
...
```

Instanciación, propiedades y fairness

```
INSTANCE inst1 =
    proc(inst2.v,TRUE,s1,s1)

INSTANCE inst2 = proc2(s1)

LTLSPEC ...
CTLSPEC ...
NORMAL_BEHAIVIOUR ...

FAIRNESS ...
COMPASSION( ... )
```

Descripción de transiciones

`[nombre]: cond_habilitación => post_condición`

La condición de habilitación es una fórmula booleana sobre el estado actual del sistema.

Las post condiciones son listas de next- valores:

$$v1' = f1,\ v2' = f2,\ v3' = f3,\ ...$$

Inyección de fallas

La inyección de fallas se realiza de modo declarativo, en la sección introducida por la palabra reservada `FAULT` de cada proctype:

```
nombre :  cond_habilitación => post_condición is TYPE
```

donde `TYPE` puede ser:

- `STOP[(trans1, trans2, ...)]`
- `BYZ([var1, var2, ...])`
- `TRANSIENT`

El lenguaje de Falluto2.0

Modelado

```
PROCTYPE machine(; send)
  VAR
    on : bool
    file : 0..9
  INIT
    !on & file = 0
  TRANS
    [send]: on
            =>
            file' = (file+1)%10
    [OnOff]: => on' = !on
ENDPROCTYPE
```

Inyección de falla

```
PROCTYPE machine(; transfer)
  VAR
    ...

  FAULT
     f1:  is STOP(OnOff)
     f2:  file = 9 => is
          BYZ(file)
     f3:  => file' = 0 is
          TRANSIENT

  INIT ...
  TRANS ...
ENDPROCTYPE
```

Especificación de propiedades

Propiedades como en NuSMV:

```
LTLSPEC q
CTLSPEC q
```

Propiedades sobre escenarios comunes

"Si solo hago buenas transiciones entonces q es verdadera":

```
NORMAL_BEHAVIOUR -> q
```

"Si eventualmente las fallas dejan de ocurrir entonces q es verdadera":

```
FINITELY_MANY_FAULTS -> q
```

Compilación de transiciones

Transiciones comunes:

`[trans]: pre => pos1, pos2, ...`

Se compila como:

$$act\#var' = trans \wedge !stop \wedge pre \wedge pos1 \wedge pos2 \wedge \\ ... \wedge unchanged$$

Donde:

$$unchanged = \bigwedge_{v \in V} (v' = v) \text{ con } V = Vars - POS - act\#var$$

Compilación de transiciones

Transiciones sincronizadas:

Por cada sincronización tendremos:

$$act\#var' = syncname \ \wedge \ !stop \ \wedge \ pre1 \ \wedge \ pre2 \ \wedge \ ... \\ \wedge \ pos1 \ \wedge \ pos2 \ \wedge \ ... \ \wedge \ unchanged$$

Donde

$$unchanged = \bigwedge_{v \in V} (v' = v) \text{ con } V = Vars - \bigcup (pos_j) - act\#var$$

stop involucra las fallas que afectan cada una de las transiciones sincronizadas.

Compilación de transiciones

Transiciones de falla:

```
fault:  pre => pos1, pos2, ...  is Type
```

Compila a:

$$act\#var' = fault \ \wedge \ !fault\_active \ \wedge \ pre \ \wedge \ fault\_active' \ \wedge \ pos1 \ \wedge \ pos2 \ \wedge \ ... \ \wedge \ unchanged$$

$!fault\_active$ y $fault\_active'$ solo en caso de fallas permanentes.

Compilación de transiciones

Transición de deadlock:

$$act\#var' = dk\#trans \ \wedge \ neg\_pre \ \wedge \ unchanged$$

$$neg\_pre \ = \ \bigwedge_{i \in instances} \ \bigwedge_{t \in transN_i} \ (\ ! \ pre_{i,t} \mid Stop_{i,t} \ )$$

$$\bigwedge_{s \in sincro} \ (\ ! \ pre_s \mid Stop_s \ )$$

Donde:

$unchanged = \bigwedge_{v \in V}(v' = v)$ con $v \in Var - act\#var$

Compilación de transiciones

Verificación de deadlock

Usamos la propiedad `CHECK_DEADLOCK`.

Es compilada como:

$$CTLSPEC\ AG\ act\#var\ !=\ dk\#trans$$

Notar relación con la transición de deadlock.

Compilación de transiciones

Compilación de la relación de transición final:

Consiste de la disyunción *exclusiva* de cada una de las transiciones del sistema:

$$\bigvee_{t \in T} t$$

Con $T =$ transiciones_comunes $\bigcup$ transiciones_sincronizadas $\bigcup$ transiciones_de_falla $\bigcup$ transicion_de_deadlock

Weak fairness entre procesos

$$F\ G\ habilitada \rightarrow G\ F\ atendida$$

Compilación:

$$FAIRNESS\ (\bigvee_{t \in T_{N_i}} act\#var = t)\ \vee\ instanceDK$$

Donde $instanceDK = \bigwedge_{t \in T_i}\ !\ pre_t$

`INST_WEAK_FAIR_DISABLE:`
deshabilita la condición de fairness entre los procesos.

Fairness con respecto a fallas

$$G\ F\ transicion\_buena$$

Compilación:

$$FAIRNESS\ (\bigvee_{t \in T_N} act\#var = t)\ \vee\ act\#var = dk\#trans$$

`FAULT_FAIR_DISABLE`:
deshabilita la condición de que a menudo se haga una transición normal.

Intérprete de trazas

```
-> State: 1.1 <-
action# = dk#action
lvar#inst#var1 = FALSE
lvar#inst#var2 = 0
ipc#inst = 0
-> State: 1.2 <-
action# = trans#inst#trans1
lvar#inst#var2 = 1
-> State: 1.3 <-
lvar#inst#var2 = 2
-> State: 1.4 <-
action# = trans#inst#trans2
lvar#inst#var1 = TRUE
ipc#inst = 1
```

```
---> State: 0 <---
inst var1 = FALSE
inst var2 = 0

@ [action] inst / trans1
---> State: 1 <---
inst var2 = 1

@ [action] inst / trans1
---> State: 2 <---
inst var2 = 2

@ [action] inst / trans2
---> State: 3 <---
inst var1 = TRUE
```

A favor

- Lenguaje simple y declarativo para la verificación de sistemas tolerantes a fallas.
- Herramienta que abstrae y automatiza la verificación de manera eficiente.
- Efectividad comprobada en casos reales de verificación (sistema de comunicación inter-satelital).

En contra

- Soporta solo una pequeña porción de las capacidades de NuSMV.
- Se debería ampliar la batería de meta-propiedades.
- El lenguaje de modelado podría hacerse más práctico en ciertos apectos.

En contra

- Soporta solo una pequeña porción de las capacidades de NuSMV.
- Se debería ampliar la batería de meta-propiedades.
- El lenguaje de modelado podría hacerse más práctico en ciertos apectos.

En realidad es trabajo a futuro ;)

Ejemplo

Cómo usar Falluto2.0

Fin

FIN