Falluto2.0
Manual de usuario.

Raul Monti

Diciembre 2012

Índice general

1 Requisitos

Para Linux:

Para usar Falluto 2.0 se precisa haber instalado de antemano NuSMV 2.5.3 (puede que funcione con versiones anteriores), y este debe ser accesible mediante la variable de entorno PATH.

También se necesita python versin 2.6.5 o superior. (puede que funcione con versiones anteriores)

2

Modelado de sistemas en Falluto2.0

En este capítulo encontrarás cómo modelar el comportamiento de un sistema en Falluto2.0, cómo especificar propiedades sobre el mismo y cómo definir restricciones sobre el proceso de verificación de de estas. Por convención los archivos de texto en donde definimos nuestros sistemas son terminados en '.fll'. (ejemplo.fll sería un nombre de archivo para verificación con Falluto2.0).

Encontrarás ejemplos de archivos '.fll' en el Apéndice A.

2.1. Proctypes

Los Proctypes definen el comportamiento de los distintos procesos intervinientes en el sistema a modelar. Cada proctype podrá ser instanciado un número arbitrario de veces para en efecto definir los procesos actuantes. Los Proctypes son delimitados con las palabras claves PROCTYPE y ENDPROCTYPE. Diagrama de un PROCTYPE:

PROCTYPE *name* (CONTEXTO ; SINCRONIZACIÓN)
PROCTYPE-BODY
ENDPROCTYPE

donde:

- *name* es el nombre del proctype (debe ser único entre los proctypes)

- CONTEXTO es una lista de variables de contexto:

  variable-de-contexto-1, ..., variable-de-contexto-n

- SINCRONIZACIÓN es una lista de nombres de acciones de sincronización:

  accion-de-sincronización-1, ..., accion-de-sincronización-m

- PROCTYPE-BODY es el cuerpo del proctype constituído por las secciones *VAR*, *INIT*, *FAULT* y *TRANS*

A continuación definimos cada sección del PROCTYPE-BODY

VAR

En esta sección declaramos las variables del proctype. Existen 3 tipos de variables en Falluto2.0:

- Variables booleanas las declaramos de esta manera:

  nombre: bool

  por ejemplo var1: bool

- Variables enteras las declaramos de esta manera:

  nombre: Entero..Entero

  por ejemplo var1: -10..5

- Variables Simbólicas las declaramos de esta manera:

  nombre: {lista de palabras y números}

  por ejemplo var1:{-1,a,b,casa,45}

FAULT

En esta sección declaramos las fallas que afectan a cada instancia del proctype. Cada falla es declarada de la siguiente manera:

nombre : [precondición] => [postcondición] is Type

donde:

- *nombre* es una palabra que define el nombre de la falla
- *precondición* es una fórmula booleana sobre el estado actual de las variables del sistema que define una condición para la habilitación de la ocurrencia de esta falla. Es decir esta transición de falla puede ocurrir si esta fórmula es verdadera en el estado actual.
- *postcondición* es una lista de next-valores indicando cambios en las variables del sistema debido a la ocurrencia de esta falla. Estos cambios quedaran plagados en el estado resultante de tomar esta transición.
- *Type* define el tipo de la falla. Hay tres tipos posibles: STOP, BYZ, TRANSIENT. Los dos primeros definen fallas de caracter permanente (solo ocurren una vez y su efecto dura hasta el 'infinito'). No así las fallas de tipo TRANSIENT que pueden ocurrir un número indeterminado de veces, y cuyo efecto es instantáneo.

Veamos cada uno de estos tipos de falla:

Las fallas de tipo **STOP** detienen transiciones definidas en el proctype. Debemos para ello identificar que transiciones deseamos que detenga. Por ejemplo:

falla: *pre* => *pos* is STOP(trans1, trans2)

una vez que ocurre deshabilita para siempre las transiciones trans1 y trans2 de la instancia correspondiente.

falla: *pre* => *pos* is STOP

una vez que ocurre deshabilita para siempre todas las transiciones de la instancia correspondiente.

Las fallas de tipo **BYZ** (Byzantine) provocan efectos bizantinos sobre variables de estado del proctype. Estos efectos se van dando a lo largo de toda la ejecución del sistema a partir de que ocurre la falla. Para ello debemos definir que variables afectan. Por ejemplo:

falla: *pre* => *pos* is BYZ(var1, var2, var9)

provoca efectos bizantinos sobre las variables var1, var2 y var9 de cada instancia cuyo proctype tenga declarada esta falla.

Por último las fallas de tipo **TRANSIENT** solo tienen el efecto definido por la post condidición que les corresponda. Por ejemplo:

falla: *pre* => var1' = FALSE, var2' = 7 is TRANSIENT

es una falla que puede ocurrir una cantidad indefinida de veces durante la ejecución del sistema, siempre que se de su precondición descripta por la fórmula booleana *pre*. Su efecto será el de cambiar el valor de var1 a FALSE y el de var2 a 7 en el estado resultante a esta transición de falla.

INIT

En esta sección definimos el estado inicial correspondiente a las instancias de este proctype. Para ello simplemente armamos una fórmula booleana que represente este estado.

TRANS

Esta sección es en la cual definimos las transiciones buenas del proctype. Cada transición se define de la siguiente manera:

[nombre]: pre => pos

Tanto nombre como pre y pos son opcionales. De nuevo *pre* es una fórmula booleana sobre el estado actual de las variables del sistema que representa la condición de habilitación para esta transición; *pos* es una lista de efectos de la transición (en forma de asignaciones a next-valores); y nombre simplemente le da un *nombre* a la transición (útil para interpretar contraejemplos, e indispensable para la sincronización).

2.2. Instansiación

Instanciamos cada proctype usando la palabra clave INSTANCE de la siguiente manera:

INSTANCE nombre = nombre-proctype(lista de parámetros)

2.2.1. Variables de contexto

Notamos que en la declaración de un proctype, podemos definir parámetros del mismo. Estos parámetros se dividen en dos secciones separadas por el símbolo ';'. En la primera sección es donde definimos parámetros para variables de contexto. Las variables de contexto son variables a las que el proctype podrá tener acceso solo en forma de lectura. En el momento de instanciación deberemos definir estos parámetros. Podemos pasar como variables de contexto en la instanciación cualquiera de los siguientes objetos:

- Valores booleanos o enteros.
- Variables de otras instancias
  (de la forma nombre-de-instancia.nombre-de-variables)
- Instancias (pasando como parámetro el nombre de las mismas)

2.2.2. Sincronización

La segunda sección en los parámetros de un proctype (la sección que esta después del símbolo ';') es utilizada para llevar a cabo la sincronización entre las distintas instancias. Toda transición dentro de la sección TRANS del proctype, cuyo nombre se corresponda con alguno de los nombres en la sección de sincronización, será sincronizada según se lo defina en la instanciación. Notar que no es posible sincronizar fallas (se supone que no decidimos sobre la ocurrencia de fallas). Para sincronizar transiciones entre 2 o mas instancias es suficiente pasar un mismo nombre de sincronización en los parámetros correspondientes a la hora de la instanciación. Por ejemplo:

```
PROCTYPE proc(;trans1, trans2)
    VAR ...
    FAULT ...
    INIT ...
    TRANS
        [trans1]
        [trans2]
        [trans2]
        ...
ENDPROCTYPE

INSTANCE inst1 = proc(sync1, sync1)
INSTANCE inst2 = proc(sync1, sync2)
```

En este caso las transiciones trans1 y trans2 de la instancia inst1 sincronizan con la transición trans1 de la instancia inst2.

2.3. Restricciones de fairness

2.3.1. Fairness y Compassion

Podemos definir restricciones de fairness incondicional (G F propiedad) de la siguiente manera:

FAIRNESS q

donde q es una formula booleana sobre el estado del sistema.

Podemos definir así también restricciones de strong fairness (G F p -> G F q) de la siguiente manera:

COMPASSION(p,q)

donde p y q son formulas booleanas sobre el estado del sistema.

2.3.2. Fairness por defecto

Por defecto Falluto trabaja sobre 2 condiciones de fairness:

- Fairness de fallas con respecto al sistema. Esta condición restringe a revisar solo aquellas trazas de ejecución donde las transiciones de falla no sean las únicas transiciones que ocurran. Osea evita aquellas situaciones en las que las fallas se ponen de acuerdo para apoderarse de la ejecución del sistema. (G F transición-buena)

- Weak fairness para instancias. Esta condición restringe a la verificación de propiedades solo sobre trazas en las que si una instancia cualquiera esta siempre habilitada para realizar una transición buena entonces siempre eventualmente sea atendida. (G habilitada -¿G F atendida)

Ambas restricciones pueden ser deshabilitadas dentro de la sección de opciones del modelado usando las palabras
$FAULT\_FAIR\_DISABLE$ e $INST\_WEAK\_FAIR\_DISABLE$ respectivamente.

2.4. Especificación de propiedades

Podemos especificar las propiedades a verificar sobre nuestro sistema utilizando Linear Time Logic (LTL) y Computing Tree Logic (CTL), de alguna de las siguientes maneras:

LTLSPEC fórmula

donde *fórmula* es una fórmula LTL válida (ver capítulo 4) sobre las trazas del sistema.

CTLSPEC formula

donde *fórmula* es una fórmula CTL válida (ver capítulo 4) sobre las trazas del sistema.

2.5. Propiedades y Fairness, sobre fallas

Dentro de la especificación de propiedades y la definición de restricciones, podemos hablar sobre la ocurrencia de las diferentes transiciones del sistema (entre ellas las fallas) usando los eventos. Un evento se describe como sigue:

$$\text{just}(nombre\_de\_la\_transicion)$$

Los eventos representan en cada estado del sistema la transición que se produjo para llegar al estado. Por ejemplo la linea 'FAIRNESS just(sync-trans1)' estaría definiendo una restricción de fairness incondicional sobre la ocurrencia de la transición llamada 'sync-trans1' (osea pide restringirse a ejecuciones en las que siempre eventualmente suceda la transición llamada 'sync-trans1')

Falluto presenta así también un conjunto de meta-propiedades para armar propiedades sobre escenarios comúnes concernientes a la ocurrencia de fallas. Estas meta-propiedades son:

- $FINITELY\_MANY\_FAULTS -> q$: para verificar si q se cumple bajo al suposición de que en cierto momento de la ejecución dejan de ocurrir fallas en el sistema.
- $FINITELY\_MANY\_FAULT(f1, f2, ..., fn) -> q$: para verificar si q se cumple bajo al suposición de que en cierto momento de la ejecución dejan de ocurrir en el sistema las fallas f1, f2, ..., fn.
- $NORMAL\_BEHAIVIOUR -> p$: para verificar si p se cumple bajo la suposición de que no ocurren fallas durante la ejecución del sistema, osea el sistema avanza solo mediante transiciones buenas.

En estas meta-propiedades p puede ser una propiedad especificada en LTL o CTL, mientras que q solo puede ser especificada en LTL.

3 Modo de uso

Uso sobre UNIX: (probado en Ubuntu)

Para usar Falluto2.0 debemos tener instalado NuSMV en tu sistema, y el mismo debe ser accesible mediante la variable de entorno PATH. Dentro del código fuente de Falluto encontramos el script python llamado Falluto2.0. Lo corremos usando python en nuestra consola, y le pasamos los siguientes parámetros:

usage: Falluto2.0 [-h] [–version] [-s path] [-co] filename

positional arguments:

filename	Input file path, where the description of the system has been written.

optional arguments:

-h, –help	show this help message and exit
–version	show program's version number and exit
-s path, –s path, -save path	save NuSMV compiled model of ths system into 'path'
-co	color output.

4

Sintáxis formal de Falluto

A continuación presentamos la sintáxis formal de Falluto2.0 en términos de Parsing Expression Grammars (PEG) y Regular Expressions (RE). Estas producciones pasan por alto los espacios en blanco, tabulaciones y saltos de linea. Consideramos aquí terminales a las letras en itálico, y a los símbolos y puntuaciones entre comillas.

Palabras reservadas de Falluto2.0:

RESERVED ⟵ *in / CHECK_DEADLOCK / OPTIONS / ENDOPTIONS / SYSNAME / just / is / FAIRNESS / COMPASSION / U / V / S / T / xor / xnor / G / X / F / H / O / Z / Y / PROCTYPE / ENDPROCTYPE / INSTANCE / TRANS / INIT / VAR / FAULT / TRUE / FALSE / AG / AX / AF / EX / EF / EG / INST_WEAK_FAIR_DISABLE / FAULT_FAIR_DISABLE / in / FINITELY_MANY_FAULT / FINITELY_MANY_FAULTS / LTLSPEC / CTLSPEC / DEFINE / FAIRNESS / COMPASSION/ NORMAL_BAHAIVIOUR*

Algunas producciones simples:

Identificadores pueden contener '.' para indicar pertenencia a un objeto (por ejemplo instancia.variable). Nombres en cambio no.

IDENT ⟵ *!* `RESERVED` *[a-zA-Z_](".")[a-zA-Z0-9_]+)?*

NAME ⟵ *!* `RESERVED` *[a-zA-Z_][a-zA-Z0..9_]**

INT ⟵ *-? ([0] / [1-9][0-9]*)*

BOOL ⟵ *TRUE / FALSE*

EVENT ⟵ *just(* `IDENT` *)*

NEXTREF ⟵ `IDENT` *'*

RANGE ⟵ `INT` *".."* `INT`

BOOLEAN ⟵ *bool*

SET ⟵ *"{"(*`IDENT` */* `INT` */* `BOOL`*) (","(*`IDENT` */* `INT` */* `BOOL`*))* "}"*

INCLUSION ⟵ `IDENT` *in (*`SET` */* `RANGE`*)*

Expresiones:

EXPRESION ⟵ `PROP`

PROP ⟵ `CONJ` *(('− >' / '< − >')* `PROP` *) ?*

CONJ ⟵ `COMP` *(('|' / '&')* `CONJ` *) ?*

COMP ⟵ `PROP` *(('<=' / '>=' / '>' / '<' / '! =' / '=')* `CONJ` *) ?*

PROD ⟵ `SUM` *(('∗' / '÷' / '%')* `PROD` *) ?*

SUM ⟵ `VALUE` *(('+' / '−')* `SUM` *) ?*

VALUE ⟵ *('('* `PROP` *')' /* `INCLUSION` */* `NEXTREF` */* `IDENT` */* `INT` */* `BOOL` */* `EVENT` */ !* `VALUE` */ -* `VALUE` *)*

NEXTLIST ⟵ `NEXTASSIGN` *(','* `NEXTASSIGN`*)**

NEXTASSIGN ⟵ `NEXTREF` *(* ′ =′ `EXPRESION` */* ”*in*” *(* `SET` */* `RANGE` *))*

Un sistema falluto se modela con un encabezado de opciones de configuración (opcional) y una serie de objetos dentro de la lista – PROCTYPE, DEFINE, INSTANCE, SPEC, CONTRAINT –:

SYSTEM ⟵ OPTIONS ? (DEFINE / PROCTYPE / INSTANCE / SPECIFICATION / CONTRAINT) *

OPTIONS ⟵ *OPTIONS* (SYSNAME [a-z0-9A-Z_.]* / *CHECK_DEADLOCK* / *FAULT_FAIR_DISABLE* / *INST_WEAK_FAIR_DISABLE*)* *ENDOPTIONS*

DEFINE ⟵ *DEFINE* IDENT ” := ” EXPRESION

PROCTYPE ⟵ *PROCTYPE* IDENT ’(’ CTXVARS ? SYNCACTS ? ’)’ PROCTYPEBODY *ENDPROCTYPE*

CTXVARS ⟵ IDENT (’,’ IDENT)*

SYNCACTS ⟵ ’;’ IDENT (’,’ IDENT)*

PROCTYPEBODY ⟵ VAR ? FAULT ? INIT ? TRANS ?

VAR ⟵ *VAR* VARDECL*

VARDECL ⟵ IDENT ’:’ (BOOLEAN / SET / RANGE)

FAULT ⟵ *FAULT* FAULTDECL*

FAULTDECL ⟵ NAME ':' (EXPRESION ? " => " NEXTEXPR ?) ?
is (BYZ / STOP / TRANSIENT)

BYZ ⟵ *BYZ* '(' IDENT (',' IDENT)*, ')'

TRANSIENT ⟵ *TRANSIENT*

STOP ⟵ *STOP* ('(' IDENT, (',' IDENT)* ')') ?

INIT ⟵ *INIT* EXPRESION ?

TRANS ⟵ *TRANS* TRANSDECL*

TRANSDECL ⟵ '[' NAME? ']' ':' EXPRESION? (" => " NEXTEXPR)?

Para la instanciación tenemos las siguientes reglas:

INSTANCE ⟵ *INSTANCE* NAME '=' NAME '(' INSTPARAMS ')'

INSTPARAMS ⟵ ((IDENT / INT / BOOL) (',' (IDENT / INT / BOOL))*) ?

Usamos las siguientes reglas para especificar las propiedades a verificar sobre el sistema modelado:

SPEC ⟵ CTLSPEC / LTLSPEC / NORMALBEHAIVIOUR / FINMANYFAULTS / FINMANYFAULT

CTLSPEC ⟵ *CTLSPEC* CTLEXP

CTLEXP ⟵ CTLVALUE CTLBINOP CTLEXP /
(*A* / *E*) '[' CTLEXP *U* CTLEXP ']' /
CTLVALUE

CTLBINOP ⟵ '&' / '|' / *xor* / *xnor* / "− >" / "< − >"

CTLVALUE ⟵ CTLUNOP CTLEXP /
'(' CTLEXP ')' /
EXPRESION

CTLUNOP ⟵ *'!' / EG / EX / EF / AG / AX / AF*

LTLSPEC ⟵ *LTLSPEC* LTLEXP

LTLEXP ⟵ LTLBOP / LTLUOP

LTLBOP ⟵ LTLUOP LTLBINOPS LTLEXP

LTLUOP ⟵ LTLUNOPS* LTLVAL

LTLVAL ⟵ EXPRESION / '(' LTLEXP ')'

LTLUNOPS ⟵ ! / *G* / *X* / *F* / *H* / *O* / *Z* / *Y*

LTLBINOPS ⟵ *U* / *V* / *S* / *T* / *xor* / *xnor* / *'|'* / *'&'* / "< − >" / "− >"

Meta-propiedades de Falluto2.0:

NORMALBEHAIVIOUR $\longleftarrow$ *NORMAL_BEHAIVIOUR* "$->$" (CTLEXP / LTLEXP)

FINMANYFAULTS $\longleftarrow$ *FINITELY_MANY_FAULTS* "$->$" LTLEXP

FINMANYFAULT $\longleftarrow$ *FINITELY_MANY_FAULT* '(' IDENT (',' IDENT)* ')' "$->$" LTLEXP

Reglas para especificación de restricciones de fairness:

CONTRAINT $\longleftarrow$ FAIRNESS / COMPASSION

FAIRNESS $\longleftarrow$ *FAIRNESS* EXPRESION

COMPASSION $\longleftarrow$ *COMPASSION* '(' EXPRESION ',' EXPRESION ')'

Apéndices

A

Ejemplos de modelado

EJEMPLO 1:

```
-----------------------------------------------------------------------

-- ARCHIVO DE EJEMPLO PARA FALLUTO 2.0
-- CONTIENE GRAN PARTE DE LA SINTAXIS DE FALLUTO EJEMPLIFICADA

-- ESTO ES UN COMENTARIO

OPTIONS
-- ACA ENCONTRAMOS ALGUNAS OPCIONES DE CONFIGURACION DEL SISTEMA

SYSNAME sitema_de_ejemplo
CHECK_DEADLOCK
FAULT_FAIR_DISABLE
INST_WEAK_FAIR_DISABLE

ENDOPTIONS

PROCTYPE proctype1( contextvar1, contextvar2; synchroact1, synchroact2)

    VAR
```

```
        var1: bool
        var2: -1..8
        var3: { symb1, symb2, symb3 }

    FAULT
        fault1: var1 => var1' = FALSE is TRANSIENT
        fault2: var2 > 2 & !var1 => is STOP
        fault3: => is STOP(synchroact1)
        fault4: var3 in {symb2, symb3, FALSE} => is BYZ(var2)

    INIT
        var1 = (TRUE | 0 > var2) & var2 in 0..5 & var3 = symb3

    TRANS
        [synchroact1]:
        [synchroact2]:
        [synchroact2]:
        []: contextvar1 != contextvar2 =>
            var3' = symb3, var1' = !var1 | contextvar2 != 7
        []: ! contextvar1 in 1..2
        [trans1]:
        [trans1]:
        [trans2]:

ENDPROCTYPE

-- A VERY SIMPLE PROCTYPE
PROCTYPE proctype2(;sync)

ENDPROCTYPE


-- INSTANTIATION

INSTANCE instance1 = proctype1(5, instance2.var2, s1, s2)
INSTANCE instance2 = proctype1(instance2.var2, 9, s2, s2)
INSTANCE instance3 = proctype2(s3)
INSTANCE instance4 = proctype2(s1)
```

```
-- PROPERTIES TO CHECK

LTLSPEC G TRUE

CTLSPEC AG TRUE

NORMAL_BEHAIVIOUR -> EG TRUE

FINITELY_MANY_FAULTS -> G TRUE

FINITELY_MANY_FAULT(instance1.fault1) -> G TRUE

-----------------------------------------------------------------------
```

EJEMPLO 2:

```
-----------------------------------------------------------------------
-- COMMIT ATOMICO DE 3 FASES


OPTIONS
SYSNAME commitAtomico
ENDOPTIONS


PROCTYPE RegularVoter(coo, v0, v1, v2)

    VAR
        phase: 0..2
        d: bool
        up: bool

    FAULT
```

```
        crash: is STOP

    INIT
        phase = 0 & up = TRUE

    TRANS
        [vote]: phase = 0 & coo.up & coo.phase = 1 =>
                d' in {TRUE, FALSE}, phase' = 1
        [abort]: phase = 0 & !coo.up => phase' = 2, d' = FALSE
        [decide0]: phase < 2 & coo.phase = 2 => phase' = 2, d' = coo.d
        [decide1]: phase < 2 & v0.phase = 2 => phase' = 2, d' = v0.d
        [decide2]: phase < 2 & v1.phase = 2 => phase' = 2, d' = v1.d
        [decide3]: phase < 2 & v2.phase = 2 => phase' = 2, d' = v2.d
ENDPROCTYPE



PROCTYPE Coordinator(v0, v1, v2, v3)

    VAR
        phase   : 0..2
        d       : bool
        up      : bool


    FAULT
        crash: is STOP

    INIT
        phase = 0 & up = TRUE

    TRANS
        [vote]: phase = 0 => phase' = 1, d' in { TRUE, FALSE }
        [decide0]: phase = 1
                & v0.up & v0.d & v0.phase = 1
                & v1.up & v1.d & v1.phase = 1
                & v2.up & v2.d & v2.phase = 1
                & v3.up & v3.d & v3.phase = 1
```

```
                => phase' = 2, d' = TRUE
        [decide1]: phase = 1
                | (!v0.up | (v0.phase >= 1 & !v0.d))
                | (!v1.up | (v1.phase >= 1 & !v1.d))
                | (!v2.up | (v2.phase >= 1 & !v2.d))
                | (!v3.up | (v3.phase >= 1 & !v3.d))
                   => phase' = 2, d' = FALSE
ENDPROCTYPE



INSTANCE coord = Coordinator(voter0, voter1, voter2, voter3)
INSTANCE voter0 = RegularVoter(coord, voter1, voter2, voter3)
INSTANCE voter1 = RegularVoter(coord, voter0, voter2, voter3)
INSTANCE voter2 = RegularVoter(coord, voter0, voter1, voter3)
INSTANCE voter3 = RegularVoter(coord, voter0, voter2, voter1)



LTLSPEC G ((coord.phase = 2 & coord.d -> (voter0.phase != 0 & voter0.d &
            voter1.phase != 0 & voter1.d & voter2.phase != 0 & voter2.d &
            voter3.phase != 0 & voter3.d)))
          &
          ((coord.phase = 0 | coord.phase = 1 | (coord.phase = 2 & !coord.d)) ->
          (((voter0.phase != 2 | !voter0.d) & (voter1.phase != 2 | !voter1.d) &
          (voter2.phase != 2 | !voter2.d) & (voter3.phase != 2 | !voter3.d) )))

-----------------------------------------------------------------------
```