Introduction à NuSMV

Cătălin Dima

LACL, Université Paris-Est Créteil

Le langage de modélisation de NuSMV

- Structures de données : types de base, tableaux, listes, structs, etc.
- Modules.
- Composition parallèle.
- Calcul concurrent.
- Composantes sous-spécifiées : nondéterminisme.
  - Les parties du système qui ne sont pas essentielles à un certain stade de conception doivent être modélisées.
  - Pour être sûr qu'il il n'y aura pas d'erreur de modélisation, on assume un comportement chaotique.
- Les langages de modélisation ne sont pas censés être exécutés :
- La plupart du temps, un modèle peut engendrer plusieurs comportements.
- Langage conçu avec l'idée que tout modèle doit être compilable en automates finis.
  - États = valuations des variables.
  - Relation de transition spécifiée à l'aide de formules "logico-mathématiques".
  - États initiaux décrits à l'aide des formules/

Le langage de modélisation de NuSMV

- Structures de données : types de base, tableaux, listes, structs, etc.
- Modules.
- Composition parallèle.
- Calcul concurrent.
- Composantes sous-spécifiées : nondéterminisme.
  - Les parties du système qui ne sont pas essentielles à un certain stade de conception doivent être modélisées.
  - Pour être sûr qu'il il n'y aura pas d'erreur de modélisation, on assume un comportement chaotique.
- Les langages de modélisation ne sont pas censés être exécutés :
- La plupart du temps, un modèle peut engendrer plusieurs comportements.
- Langage conçu avec l'idée que tout modèle doit être compilable en automates finis.
  - États = valuations des variables.
  - Relation de transition spécifiée à l'aide de formules "logico-mathématiques".
  - États initiaux décrits à l'aide des formules/

Le premier programme NuSMV

Première variante !

```
MODULE main
VAR
 b0 : boolean;
ASSIGN
 init(b0) := 0;
 next(b0) := !b0;
```

Un module NuSMV comprend :

- Des déclarations de variables.
- Des affectations des valeurs initiales aux variables.
- Des règles décrivant comment les variables évoluent au cours de chaque comportement.

Types de données et déclarations de variables

- Déclarations : précédées du mot-clé VAR.
- Booléens.

  VAR
  ```
    s: boolean;
  ```
- Enumeratifs :

  ```
    st : {ready, busy, waiting, stopped} ;
  ```
- Entiers bornés :

  ```
    n : 1..8;
  ```
- Mots (tableaux de bits) :

  ```
    w : unsigned word[5];
  ```
- Tableaux typés :

  ```
    a : array 1..10 of signed word[10];
    b : array -10..200 of array 22..25 of {OK, start, stop};
  ```

Déclarations d'états initiaux

```
init(<variable>) := <simple_expression> ;
```

- `simple_expression` et `variable` sont typés et doivent voir leurs types correspondre.
- Operateurs :
  - Operateurs arithmétiques : `+, -, *, /, mod` (unaire).
  - Comparaisons : `<, >, <=, >=, =, !=`.
  - Logique : `!, &, |, xor, ->, <->`.
  - Operateurs ensemblistes : `in, union` et énumération `{premier, deuxieme, 2345, c, blabla}`.
- Opérateur d'indice de tableau : `a[15], a[b]`.
- Autres opérateurs (plus tard).

Convention

Si la valeur initiale d'une variable n'est pas spécifiée, il est assumé que la variable respective est initialisée avec n'importe quelle valeur.

Affectations nondéterministes

- Opérateurs qui construisent des ensembles de valeurs : `range constant, union`.
- Lorsque l'opérande droit d'une affectation représente un tel ensemble, l'identifiant à gauche de l'affectation prend une valeur quelconque dans l'ensemble respectif de manière nondéterministe.
- Pas de possibilité de construire des ensembles d'ensembles !

Expressions de branchement

- Opérateur `case` :

```
case
   val1 : expr1;
   val2 : expr2;
   ...
   TRUE : exprn;
esac
```

  - Les opérandes `val`*i* peuvent être des ensembles.
  - L'opérateur produit une valeur : la valeur de la première expression pour laquelle l'égalité avec la valeur correspondante est vraie.
  - Le `TRUE` final est là car les variantes doivent être exhaustives.

- Opérateur ternaire `if-then-else` :

```
cond_expr ? basic_expr1 : basic_expr2
```

  - La valeur renvoyée est celle de la `basic_expr` qui correspond à la valeur vrai de la `cond_expr`.

Déclarer la relation de transition

- L'opérateur `next` :

  ```
  next(variable) := next_expression ;
  ```

- Le typage de `variable` et de `next_expression` doit être identique (modulo l'utilisation des expressions ensemblistes dans la dernière).
- Il est permis d'utiliser d'autres expressions `next` dans la `next_expression` (mais avec des restrictions !) :

  ```
  next(a) := {a, a+1};
  next(b) := b + (next(a) - a) ;
  ```

- Si aucune affectation de type `next` n'est pas spécifiée pour une variable v, alors il est assumé que cette variable évolue de manière non-déterministe à chaque pas de la relation de transition !

Déclarer un système de transition à l'aide de `init` et `next`

```
MODULE main
VAR
 b0 : boolean;
 b1 : boolean;
ASSIGN
 init(b0) := 0;
 next(b0) := !b0;
 init(b1) := 0;
 next(b1) := ((!b0 & b1) | (b0 & !b1));
```

Dessiner l'automate fini pour le modèle décrit !

Règles pour les affectations

- Les affectations pour les valeurs courantes des variables sont permises (mais avec des restrictions !) :

```
MODULE main
VAR
 b0 : boolean;
 b1 : boolean;
 out : 0..3;
ASSIGN
 init(b0) := 0;
 next(b0) := !b0;
 init(b1) := 0;
 next(b1) := ((!b0 & b1) | (b0 & !b1));
 out := b0 + 2*b1;
```

Règles pour ASSIGN

- Les affectations décrivent des systèmes d'équations qui précisent comment l'automate fini évolue dans le temps.
- Restrictions sur la construction du système d'équations :
  - Règle de l'affectation unique : toute variable doit être la cible d'une seule affectation (ou d'aucune).
  - Règle des dépendances circulaires : chaque *circuit* dans le système d'équations doit comprende au moins une affectation de type `next`.
- Jeux d'affectations illégaux :

```
x := y;  y := x;
x := 2; x := 3;
x := 14; next(x) := x+1;
next(x) := x & next(x);
```

- Exemples corrects :

```
x := y; next(y) := x;
next(x) := x & next(y); next(y) := y & x;
```

Déclaration de relation de transition avec TRANS

- Mot-clé TRANS.
- Expression booléenne détaillant les correspondances entre valeurs de variables avant et après la transition.
- Peut ne pas contenir des affectations de type `next`.
- Plusieurs sections TRANS acceptées, la relation de transition sera la conjonction des relations décrites dans chaque section.
- Habituellement on donne des formules en forme normale disjonctive : des disjonctions de conjonctions.
- Exemples :
  - La valeur de `output` est, après chaque transition, soit la valeur du `input`, soit une copie de la valeur précédente du `output` :

    ```
    next(output) = !input | next(output) = output
    ```
  - Après chaque transition, une seule des deux variables booléennes `var1` et `var2` peut être vraie :

    ```
    ! (next(var1) & next(var2))
    ```

Déclarations d'invariants

- Mot-clé `INVAR` :

```
INVAR (state1 = critical & state2 = critical)
```

- Si une variable doit ne pas changer de valeur :

```
ASSIGN a := x;
INVAR a=x;
```

- Variables "gelées" (ou déclaration de constantes) :

```
FROZENVAR a: 0..8;
VAR d: 0..8;
ASSIGN
  init(a) = b;
```

- Pas de `next` sur variable gelée !

Modules

- Déclarations encapsulées de variables, constantes et spécifications les concernants.
- Sorte de `struct` généralisé, où on inclut aussi les contraintes reliant les membres.
- Le premier exemple : le module `main`.
- L'identificateur de module sert à référencer les variables déclarées à l'intérieur du module.
- Les paramètres du module servent à spécifier les contraintes les reliant aux variables déclarées à l'intérieur du module.
- Une instanciation du module sert la plupart du temps à définir une variable.

Exemple de module

```
MODULE main
VAR
  a: boolean;
  b: mod(a); // b.y = FALSE !
INIT a=true;
MODULE mod(x)
VAR
  y: boolean;
ASSIGN
  y:= !x;
```

La variable b n'a pas de valeur primitive !

Exemple : compteur à trois bits

```
MODULE counter(reset)
VAR
  output : 0..7;
ASSIGN
  init(output) := 0;
  next(output) :=
  case
    reset = 1 : 0;
    output < 7 : output + 1;
    output = 7 : 0;
    1 : output;
  esac;

MODULE main
VAR
  reset : boolean;
  dut : counter(reset);
ASSIGN
  init(reset) := 1;
DEFINE
  cnt_out = dut.output;
```

Simulation à l'aide de NuSMV

- `read_model -i` : lit un fichier contenant un modèle de système de transition.
- `go` : intialise le modèle pour la vérification/simulation.
- `pick_state` : choisit un état initial comme état initial de la simulation.
  - Option `-r` : choix aléatoire.
  - Option `-c "contrainte"` : choix parmi les états satisfaisant la contrainte spécifiée.
  - Option `-i` : choix interactif, marche de pair avec `-a`, qui affiche tous les états et variables gelées.
- `simulate` : lance la simulation selon les options choisies :
  - Option `-k n` : profondeur maximale du chemin construit, selon les contraintes.
  - Option `-i` : choix interactif à chaque pas, marche de pair avec `-a`.
  - Option `-r` : choisit les états de manière aléatoire.
  - Option `-c "contrainte"` : chaque choix se fait parmi les états qui satisfont la contrainte. La contrainte ne peut pas contenir de `next`.
  - Option `-t "contrainte"` : pareil que pour `-c`, mais la contrainte peut contenir des `next`.
- `reset` : pour lire un nouveau modèle.

Autres commandes (à compléter)

- `check_fsm` : détection de deadlock, vérifie que le système de transition est total.
  - Si non-total, affiche un état d'interblocage.
- `compute_reachable` : construction de l'ensemble des états atteignables.
  - Option `-k n` : limite le nombre de pas pour calculer l'espace d'états atteignable.
  - Option `-t s` : limite la durée du calcul à `s` secondes.
- `print_reachable_states` : affiche le nombre d'états atteignables ou, si demandé, les états.
  - Option `-f` : formule représentant les états atteignables.
  - Option `-o fichier` : fichier destination.
  - Option `-v` : verbeux.