Systèmes Synchrones
L. Mandel, M. Pouzet & J. Vuillemin

Projet : Lustre Model Checker

Le but de ce projet est d'écrire un model checker pour un sous-ensemble du langage synchrone Lustre. Nous appellerons ce model checker `lmoch`. Il utilisera les principes de k-induction.

Ce projet s'inspire du model checker Kind (`http://clc.cs.uiowa.edu/Kind`).

1 Principes

Le model checker `lmoch` prend en entrée un noeud Lustre qui a une sortie booléenne et doit prouver l'invariant que cette sortie est toujours vraie. Pour cela, il transforme la définition du noeud en une formule de logique du premier ordre qui encode sa sémantique, puis en utilisant un démonstrateur automatique il prouve que la sortie est vraie.

Illustrons cela sur un exemple simple. Il s'agit d'un compteur qui est incrémenté à chaque fois que son entrée `tic` est vraie. Nous voulons montrer que la valeur du compteur est croissante.

```
node incr (tic: bool) returns (ok: bool);
var cpt : int;
let
  cpt = (0 -> pre cpt) + if tic then 1 else 0;
  ok = true -> (pre cpt) <= cpt;
tel
```

Nous représentons les flots comme des fonctions des numéros d'instant dans les valeurs. Ainsi, le noeud ci-dessus peut-être représenté par la formule :

$$\Delta_{incr}(n) = \left\{ \begin{array}{lcl} cpt(n) & = & ite(n = 0,\, 0,\, cpt(n-1)) + ite(tic(n),\, 1,\, 0) \\ ok(n) & = & ite(n = 0,\, true,\, cpt(n-1) \leq cpt(n)) \end{array} \right\}$$

Il faut maintenant prouver l'invariant $\forall n \geq 0.\ ok(n)$. Cela peut se faire par induction en prouvant le cas de base et le cas inductif :

$$\Delta_{incr}(0), \Delta_{incr}(1) \models ok(0) \wedge ok(1)$$
$$\Delta_{incr}(n), \Delta_{incr}(n+1), ok(n) \models ok(n+1)$$

Dans le cas général, si on appelle $\Delta(n)$ la formule définissant un noeud à l'instant n et $P(n)$ la propriété à prouver pour toute valeur de n. La propriété est *k-inductive* si :

$$\Delta(0), \Delta(1), \ldots, \Delta(k) \models P(0) \wedge P(1) \wedge \ldots \wedge P(k)$$
$$\Delta(n), \Delta(n+1), \ldots, \Delta(n+k+1), P(n), P(n+1), \ldots, P(n+k) \models P(n+k+1)$$

2 SMT Solver : Alt-Ergo-zero

Afin de montrer qu'un invariant définit par noeud Lustre est k-inductif, nous utilisons un *SMT solver* pour prouver la formule du cas de base et la formule du cas inductif. Dans ce projet, nous vous conseillons d'utiliser Alt-Ergo-zero (`http://cubicle.lri.fr/alt-ergo-zero/`), un SMT solver qui se présente sous forme de bibliothèque OCaml.

Dans la suite de cette section, nous montrons comment utiliser Alt-Ergo-zero pour prouver que l'invariant défini par le noeud `incr` est 1-inductif.

Les formules de Alt-Ergo-zero sont exprimées dans le langage suivant :

f	::=	t	terme
		$\mid t\ cmp\ t$	comparaison entre termes
		$\mid f\ comb\ f$	opération logique entre formules
t	::=	c	constante entière, réelle ou booléenne
		$\mid t\ op\ t$	opération arithmétique entre termes
		$\mid ite(f,\ t,\ t)$	conditionnelle
		$\mid f(t, \ldots, t)$	application de fonction non interprétée
cmp	::=	$= \mid <> \mid < \mid \leq$	
$comb$	::=	$\wedge \mid \vee \mid \Rightarrow \mid \neg$	
op	::=	$+ \mid - \mid * \mid / \mid mod$	

La définition $\Delta_{incr}(n)$ peut se traduire dans ce langage en :

$$\Delta_{incr}(n) = \left\{ \begin{array}{l} cpt(n) = ite(n = 0,\ 0,\ cpt(n-1)) + ite(tic(n),\ 1,\ 0) \\ ok(n) = ite(n = 0,\ true,\ aux(n)) \\ (aux(n) \Rightarrow cpt(n-1) \leq cpt(n)) \wedge (cpt(n-1) \leq cpt(n) \Rightarrow aux(n)) \end{array} \right\}$$

qui utilise les symboles suivants :

$$tic : int \to bool \qquad cpt : int \to int \qquad ok : int \to int \qquad aux : int \to bool$$

Regardons comment coder cette définition en Alt-Ergo-zero. Il faut commencer par ouvrir la bibliothèque :

```
open Aez
open Smt
```

Ensuite, il faut déclarer les symboles utilisés dans les formules :

```
let declare_symbol name t_in t_out =
  let x = Hstring.make name in (* creation d'un symbole *)
  Symbol.declare x t_in t_out; (* declaration de son type *)
  x

let tic = declare_symbol "tic" [ Type.type_int ] Type.type_bool
let ok = declare_symbol "ok" [ Type.type_int ] Type.type_bool
let cpt = declare_symbol "cpt" [ Type.type_int ] Type.type_int
let aux = declare_symbol "aux" [ Type.type_int ] Type.type_bool
```

Nous pouvons maintenant écrire des fonctions `def_cpt`, `def_ok` et `def_aux` qui prennent en argument un terme qui représentent la valeur de l'instant n et retournent respectivement les formules qui définissent les flots *cpt*, *ok* et *aux*.

```
let def_cpt n =
  (* cpt(n) = ite(n = 0, 0, cpt(n-1)) + ite(tic(n), 1, 0) *)
  let ite1 = (* ite(n = 0, 0, cpt(n-1)) *)
    Term.make_ite
      (Formula.make_lit Formula.Eq [n; Term.make_int (Num.Int 0)])
      (Term.make_int (Num.Int 0))
      (Term.make_app cpt
         [ Term.make_arith Term.Minus n (Term.make_int (Num.Int 1)) ])
  in
  let ite2 = (* ite(tic(n), 1, 0) *)
    Term.make_ite
      (Formula.make_lit Formula.Eq [Term.make_app tic [n]; Term.t_true])
      (Term.make_int (Num.Int 1))
      (Term.make_int (Num.Int 0))
  in
  (* cpt(n) = ite1 + ite2 *)
  Formula.make_lit Formula.Eq
    [ Term.make_app cpt [n] ;
      Term.make_arith Term.Plus ite1 ite2 ]

let def_ok n =
  (* ok(n) = ite(n = 0, true, aux(n)) *)
  Formula.make_lit Formula.Eq
    [ Term.make_app ok [n] ;
      Term.make_ite
        (Formula.make_lit Formula.Eq [n; Term.make_int (Num.Int 0)])
        Term.t_true
        (Term.make_app aux [n]) ]

let def_aux n =
  let aux_n = (* aux(n) = true *)
    Formula.make_lit Formula.Eq [ Term.make_app aux [n]; Term.t_true ]
  in
  let pre_cpt_le_cpt = (* cpt(n-1) <= cpt(n) *)
    Formula.make_lit Formula.Le [ Term.make_app cpt
                                    [ Term.make_arith Term.Minus
                                        n (Term.make_int (Num.Int 1)) ];
                                  Term.make_app cpt [n] ]
  in
  Formula.make Formula.And
    [ Formula.make Formula.Imp [ aux_n; pre_cpt_le_cpt ] ;
      Formula.make Formula.Imp [ pre_cpt_le_cpt; aux_n ] ]
```

La définition Δ_{incr} et l'invariant à prouver s'écrivent :

```
let delta_incr n = Formula.make Formula.And [ def_cpt n; def_ok n; def_aux n ]
let p_incr n = Formula.make_lit Formula.Eq [ Term.make_app ok [n]; Term.t_true ]
```

La première étape est de prouver le cas de base de l'induction :

$$\Delta_{incr}(0), \Delta_{incr}(1) \models ok(0) \wedge ok(1)$$

Pour cela, on crée une instance d'un SMT solver :

```
module BMC_solver = Smt.Make(struct end)
```

On charge les hypothèses dans le SMT solver :

```
let () =
  BMC_solver.assume ~id:0 (delta_incr (Term.make_int (Num.Int 0)));
  BMC_solver.assume ~id:0 (delta_incr (Term.make_int (Num.Int 1)));
  BMC_solver.check()
```

On demande au solver si le cas de base est vrai :

```
let base =
  BMC_solver.entails ~id:0
    (Formula.make Formula.And [ p_incr (Term.make_int (Num.Int 0)) ;
                                p_incr ((Term.make_int (Num.Int 1))) ])
```

On fait de même pour le cas inductif :

$$\Delta_{incr}(n), \Delta_{incr}(n+1), ok(n) \models ok(n+1)$$

```
module IND_solver = Smt.Make(struct end)

let ind =
  let n = Term.make_app (declare_symbol "n" [] Type.type_int) [] in
  let n_plus_one = Term.make_arith Term.Plus n (Term.make_int (Num.Int 1)) in
  IND_solver.assume ~id:0 (Formula.make_lit Formula.Le [Term.make_int (Num.Int 0); n]);
  IND_solver.assume ~id:0 (delta_incr n);
  IND_solver.assume ~id:0 (delta_incr n_plus_one);
  IND_solver.assume ~id:0 (p_incr n);
  IND_solver.check();
  IND_solver.entails ~id:0 (p_incr n_plus_one)
```

À partir des valeurs de `base` et `ind`, on peut conclure sur la validité de l'invariant :

```
let () =
  if not base then Format.printf "FALSE PROPERTY"
  else if ind then Format.printf "TRUE PROPERTY"
  else Format.printf "Don't know"
```

Le fichier contenant la preuve est disponible sur `http://www.lri.fr/~mandel/mpri/projet/incr_proof.ml`. Il peut être compiler et exécuter :

```
--> ocamlopt -o incr_proof unix.cmxa nums.cmxa aez.cmxa incr_proof.ml
--> ./incr_proof
TRUE PROPERTY
```

3 Travail demandé

Nous vous fournissions un parseur et un typeur pour un sous-ensemble du langage Lustre :

`http://www.lri.fr/~mandel/mpri/projet/lmoch.tar.gz`

C'est un sous-ensemble dans lequel, en particulier, il n'y a pas d'horloges et les données sont uniquement de type `int`, `real` ou `bool`.

Le but du TP est de réaliser un model checker pour ce langage. Vous devrez lire (et comprendre) les articles suivants :

— Mary Sheeran and Satman Singh and Gunnar Stalmark. Checking safety properties using induction and a SAT-solver. FMCAD 2000.

— George Hagen and Cesare Tinelli. Scaling up the formal verification of Lustre programs with SMT-based techniques. FMCAD 2008.

— George Hagen. Verifying safety properties of Lustre programs : an SMT-based approach. PhD dissertation 2008.

D'un point de vue implantation, à partir de programmes Lustre typés (la structure de donnée définie dans le fichier `typed_ast.ml`) vous allez devoir générer des formules dans la logique de Alt-Ergo-zero et écrire un moteur de model checking.

Le projet est à faire seul ou en binôme. Il doit être remis par email à `louis.mandel@college-de-france.fr` et `marc.pouzet@ens.fr` avant le **1er décembre 23h59**. Votre projet doit se présenter sous forme d'une archive `tar` compressée, appelée `vos_noms.tgz` qui doit contenir un répertoire appelé `vos_noms` (exemple : `dupont-durand.tgz`). Cette archive doit contenir le code du projet, des exemples et un rapport de deux ou trois pages expliquant les choix techniques, les extensions réalisées, les limitations du projet, etc. Le rendu du projet sera suivi d'une soutenance.