Rapport

Stéphane Romany
Bousalem Amina

Abstract

Ce projet a pour but créer un model checker pour une version simplifiée du langage synchrone Lustre en se basant sur la méthode de k-induction afin de pouvoir simuler la sémantique temporelle du langage. Le code de ce projet est écrit en Ocaml, et le solver SMT utilisé est celui de la lib Alt-ergo-zero. Nous expliciterons les divers étapes du codage.

1 Compilation

Pour lancer le programme ,il faut taper la commande ./lmoch ..examples/fichier.lus le nom du noeud à vérifier. On peut rajouter plusieurs options comme -ind pour déterminer k , time pour savoir le temps pris par le solveur pour vérifier le noeud ...

2 Transformation

`De l'ast typé a l'aez...`

2.1 Convertion

Le but principal de cette étape est de convertir les expressions, afin de séparer les différents types d'opérations: Opérations arithmétiques, Comparaisons et Combinaisons booléennes et Branchements conditionels.
De plus, on effectue une étape de transformation de tous les opérateurs Ge et Gt (resp. Greater equal, Greater than) en opétateur Le et Lt (resp. Less equal, Less than) afin de pouvoir coller à l'interface de Alt-ergo-Zero. En effet celui ci n'implémente pas d'opérateurs Ge, Gt.
Lors de cette conversion l'objet retourné est une z_expression. La fonction convert se charge de reconstruire les z_equations en s'inspirant du même schéma que les t_equations reçus en entrée.
La fonction propagate_convert se charge de transformer récursivement les autres expressions dans l'ast typé.

```
(* Code *)
let convert (node: z_node) (eq: TE.t_equation) : z_equation =
   let z_expr =
      { zexpr_desc=propagate_convert eq.teq_expr;
          zexpr_type=eq.teq_expr.texpr_type;
      }
in
  { zeq_patt=eq.teq_patt;
    zeq_expr=z_expr;
    }
```

2.2 Normalisation

L'étape de normalisation sert à convertir les expression de types Comparaisons ou Combinaisons en termes auxiliaires afin de pouvoir matcher avec l'interface d'Aez.
Comme indiqué dans l'énoncé : la fonction normalize génère un identifiant et déclare un nouveau symbole du mme nom, "auxi" (Avec i, un identifiant pour s'assurer qu'il n'y aura pas deux variable déclarer avec le meme symbole, et i = 0 ... n).
Puis cet identifiant est intégré à la liste d'équation du noeuds en tant que nouveau pattern associer à l'équation qu'il représente. Ce symbole auxilliaire servira a construire les formules de la forme :

$$(aux(n) \Rightarrow patt1(n-1) \leq patt2(n)) \wedge (patt1(n-1) \leq patt(n) \Rightarrow aux(n)) \tag{1}$$

3 Construction des formules

3.1 Glaçage

Utilisation de glaçons, fonctions dont l'exécution est retardé afin de construire les formules au moment de l'appel du SMT solver. Nous avons choisi cette stragégie d'implementation car ce n'est que lors de l'appel du solver que la valeur de n est connue. Ici une fonction se chargera de créer les formules en prennant le paramètre n qui lui sera fourni plus tardivement.

```
(* Code *)
let build_formula node patt_ty expr =
  let var_symbol = fst patt_ty in
  (* Freeze : Ici une fonction retournee en tant que valeur.*)
  (fun (n: Term.t) -> make_formula (!id_node) node var_symbol expr.zexpr_desc n)
```

4 K-induction

Cette partie implémente le solver par k-induction, elle contient la fonction check qui prend un noeud et vérifie la propriété P pour les deux cas *base* et *induction* avec un certain k qu'on choisit (via l'option -ind <value> du programme lmoch.
Cette propriété est exprimée par la ou les variable de sorties du noeud, de type booléennes.Dans le cas de plusieurs sorties La formule crée sera la conjonction entre les différentes propriété P concernant le sorties du noeud. Si le cas de base est faux, le solver s'arrête et affiche "FALSE_PROPERTY", sinon on vérifie le cas inductif. S'il peut être résolu alors on a atteint la satisfiabilité du modèle et on affiche "TRUE_PROPERTY" sinon on affiche "UNKNOWN_PROPERTY".

Optimisation

Dans l'état actuel de l'implémentation, nous avons tenté une petite optimisation citée dans la thèse de Georges Hagen : Compression de chemins.
Cette optimisation permet de renforcer les assumptions et d'éliminer certain chemins qui mènent à des configurations équivalentes.
On a ajouté dans le cas inductif une assumption. On a choisi deux variables i et j tel que $i < j < n$ pour chercher une certaine variable var qui peut être dans la liste des inputs ou locals tel que $var(i) \neq var(j)$.

Une autre idée d'optimisation serait de supprimer des formules toutes les variables locales ou d'entrées qui n'ont aucuns liens avec le résultat du noeud permettant ainsi d'alléger les formules.

5 Conclusion

Ce projet nous a permis de découvrir de nombreuses fonctionnalités du langage Ocaml.
Nous avons également pu implémenté une grande partie de la procédure de vérification, bien que nous ayons eu un peu de mal au début à comprendre le code fournit, et à réfléchir sur la construction des formules (qui nécessitaient un paramètre connu uniquement au moment de la vérification) .
A cause d'un choix d'implémentation qui avait l'air judicieux au départ, nous ne pouvons pas gérer les cas des appels de noeuds par le noeuds principal. Néanmoins nous avons identifier le problème, nécessitant un peu plus de temps pour le corriger.