Un langage impératif pour la description des systèmes indéterministes/stochastiques

Vérimag/grenoble

Motivations

- initialement le test (description des environnements),
- plus généralement pour la simulation, le prototypage ...

Dans ALIDECS :

Description de la fonctionnalité des composants “non programmés”

- à l’état de spéc.
- modélisation de composants matériels
- modélisation d’environnements physiques
- description de contrats temporels complexes

Principes de base

Variables

Un système réactif dont les variables sont connues :

- **entrées (non contrôlables) E**
- **sorties (contrôlables) S (+ éventuellement des variables internes V)**

Il obéit au modèle synchrone :

- **fonctionnement sur une horloge discrète implicite,**
- **les S et les V dépendent de la valeur courante des E, mais aussi des valeurs à l'instant précédent (les "pres" de E, S, V)**

Réactions

Les réactions sont indéterministes :

- **les S et V ne sont pas *fonctions* des E et des pre, juste en *relation***
- **une réaction indéterministe (i.e. un instant dans l'exécution du système) peut être représentée par une expression booléenne sur les variables et leurs `pre`.**
- **On parlera de *contrainte***

Enchaînement des contraintes

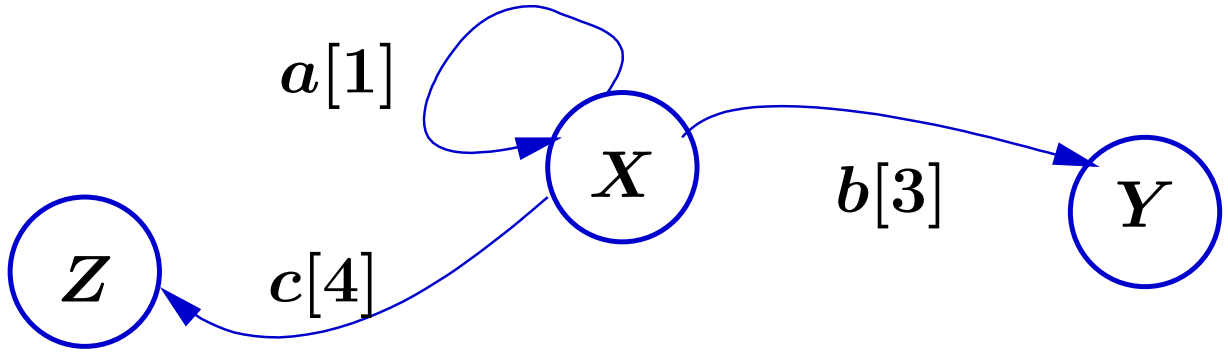
N.B. le fait d'avoir les `pre` suffit pour exprimer la dynamique d'un système mais ce n'est pas pratique :

- **difficile d'exprimer des comportements séquentiels (type scénarios)**
- **difficile d'introduire des notions probabilistes**

Modèle des automates à poids

- **structure de contrôle explicite (états/transitions)**
- **les transitions sont étiquetées par une contrainte**
- **et par un *poids relatif***

Exemple



Depuis X, une transition est choisie parmi les *satisfiables* selon le rapport de leurs poids. Par exemple :

- **a, b et c satisfiables : $P_a = 1/7$, $P_b = 3/7$ et $P_c = 4/7$.**
- **seulement a et c satisfiables : $P_a = 1/5$, $P_b = 0$ et $P_c = 4/5$.**
- **seulement a : elle est choisie avec la proba 1**

N.B. si aucune transition n'est satisfiable, le système se bloque !

Extensions du modèle

- **Les poids *dynamiques*, fonction des non-contrôlables *uniquement*, permettent d'exprimer des comportements "vivants"**
- **Le poids *infini* (resp. *infime*) comme formalisation du choix impératif (resp. de dernier recours).**
- **Transitions "branchues", pour exprimer des probabilités complexes**

Langage de haut niveau ?

- **les automates à poids sont un *modèle*,**
- **besoin d'un *langage* de plus haut niveau**

Lutin v1

- Même principes de base (variables, contraintes)
- Structures de contrôle basée sur des opérateurs réguliers :
  - ⋆ la séquence : `t fby t'`
  - ⋆ la boucle : `loop{ t }` et ses versions contraintes :
    `loop~200:30 { t }`
    `loop[100:400] { t }`
  - ⋆ le choix stockastique : { t1 `weight` w1 `|` ... `|` tn `weight` wn }
  - ⋆ le choix prioritaire : { t1 `|>` ... `|>` tn }
  - ⋆ la projection : `assert` c `in` t

Problème essentiel : qu'elle est la *bonne* notion de modularité ?

Lutin V2

- La notion de système indéterministe n'est pas la bonne "brique" de base pour une approche modulaire.
- La notion retenue est celle de "bouts de comportements", réutilisables dans le contexte de n'importe quel système.

  Par exemple :

```
let increase(x :  real) = loop { x >= pre x }
```

- Plus généralement : on rajoute une "couche" fonctionnelle à Lutin, *sans ordre suppérieur ni récursivité* (on reste "temps-réel").
- Un calcul de type simple permet d'assurer la correction des programmes

Système de type

- Les types de valeurs prédéfinisi `int`, `bool` et `real`
- Le “type” `trace` comme abstraction de la notion de comportement dynamique
- Toute expression `bool` est implicitement une contrainte, donc une `trace` (de longueur 1)
- Les opérateurs (y compris les structures de contrôle) sont typées, par exemple :
  - $\star$ $\texttt{fby} : \texttt{trace} \times \texttt{trace} \mapsto \texttt{trace}$
  - $\star$ $\texttt{weight} : (\texttt{trace} \times \texttt{int})^{+} \mapsto \texttt{trace}$

Syntaxe abstraite

- system ::= `node` id (varlist) `returns` (varlist) = exp
- exp ::= id | constant | id (explist)
  | `let` id : type (params) = exp `in` exp
  | `exist` varlist `in` exp
  | statement
- statement : `fby`, `loop`, choix etc.

  + quelques constructions inspirées par les “faiblesses” de la v1
- n.b. les opérateurs arithmétiques et logiques sont des fonctions prédéfinies

Exceptions

- L'impossibilité de gérer les blocages est un problème : besoin d'un "catch"
- Plus généralement, introduire une notion d'exception pour gérer les "passages" de contrôle transversaux
- Définition d'une exception, au (top-level, pas de contexte) :
  `exception` Ident
- Une exception prédéfinie (mot-clé) : `Deadlock`
- Levée : `raise` Id
  (type : `exception` $\mapsto$ `trace`)
- Interception : `try` exp `with` Id `do` exp
  (type : `trace` $\times$ `exception` $\times$ `trace` $\mapsto$ `trace`)

Concurrence

(en cours de réflexion)

- **Réel besoin, mais difficile à définir et maîtriser (quid des probas ? quid de l'arrêt ?)**
- **Une piste : { exp1 || exp2 } où :**
  - ⋆ **termine si quand le dernier termine (à la Esterel)**
  - ⋆ **résolution des probas par choix à pile ou face du premier "joueur"**
  - ⋆ **alternative : { exp1 &| exp2 }, les poids de exp1 sont traités parfaitement, exp2 est "servi" après**

Automates

(en cours de réflexion)

Il “suffit” d’accepter la récursion terminale pour la définition des trace :

```
let rec X = { a bfy X | b fby Y }
and     Y = { c bfy X | d fby Z }
and     Z = { e bfy Y | f fby Z }
in A
```

Est-ce la bonne solution ?

Avancement

- Le langage est implémenté (sauf concurrence et automates)
- Le compilateur est partiellement écrit :
  - ★ typage “grossier” et portée des ident
  - ★ typage “fin” et expansion
  - ★ reste à faire la génération d’automate