Défense et illustration des algèbres de processus

Hubert GARAVEL

projet VASY

INRIA Rhône-Alpes

655, avenue de l'Europe

38330 Montbonnot Saint Martin

France

Plan

- Modélisation des systèmes asynchrones
- Quatre critères essentiels
- Modélisation des données
- Modélisation des comportements
- Conclusions

Systèmes asynchrones

- Définition : Ensemble de processus
  - pouvant s'exécuter à des vitesses différentes
  - n'ayant pas forcément de mémoire commune

- Nombreuses applications :
  - télécommunications (protocoles…)
  - logiciel (systèmes distribués…)
  - matériel (multiprocesseurs, circuits asynchrones)

Plusieurs défis à relever

- Les systèmes asynchrones sont difficiles à concevoir
- Complexification croissante des applications
- Besoins de sécurité élevés

Besoin de langages de modélisation

- Les langages de programmation usuels (C/C++, Java…) sont de trop bas niveau
- Des langages de plus haut niveau sont indispensables
- Importance des outils de preuve/vérification

De multiples propositions…

- Modèles théoriques (origine: recherche)
  - réseaux de Petri
  - automates communicants
- Algèbres de processus (origine: recherche)
  - CCS [Milner]
  - CSP [Hoare]
  - LOTOS, E-LOTOS (normes ISO), …
- Langages dédiés (origine : industrie)
  - SDL (télécommunications)
  - UML (méthodes à objets)
  - systemC (conception de matériel)
  - …

Comment comparer ces diverses propositions ?
A quoi ressemblerait un langage "idéal" ?

Quatre critères essentiels

1er critère: Expressivité

- Pouvoir modéliser les caractéristiques essentielles des systèmes asynchrones

- Puissance d'expression : ne pas être limité par des impossibilités techniques

- Simplicité : éviter les "contorsions" inutiles
  - Complexité => coût (formation, outillage)
  - Complexité => échec

  Une certaine complexité reste inévitable

2ème critère : Universalité

- Convenir à tous les domaines d'activité où le parallélisme asynchrone intervient

- Ne pas être trop spécifique à un secteur particulier :

  => réduction des coûts (outils, formation)

  => gage de pérennité

- Les "couches métiers" sont possibles, mais en périphérie (interface utilisateur)

3ème critère : "Exécutabilité"

Pourquoi modéliser les systèmes ?

1. Préciser les spécifications
2. Détecter les erreurs (~70%)
3. Servir de documentation

Permettre la génération de code (efficace)

pour aller plus loin :

- mise au point / simulation
- prototypage rapide
- génération de tests

4ème critère : "Vérifiabilité"

- "*Correct by design*" reste difficile
- Besoin d'outils de vérification/preuve
- Impact fort sur le choix du langage
  - preuve => sémantique formelle
  - "model checking" => abstraction, compositionnalité
- Continuum entre vérification et implantation

  What you prove is what you execute

Structure d'un langage de modélisation

Un langage complet doit permettre :

- la modélisation des données
  (types, variables, expressions, fonctions…)
- la modélisation des comportements
  (processus, signaux, temps réel…)
- avec des moyens de structuration
  (modules, classes, bibliothèques…)

Modélisation des données

Cinq approches tentées

- Réseaux de Petri avec jetons colorés
- Importation de types et fonctions externes

  (Esterel)
- Sous-ensemble de langages impératifs

  (Pascal pour Estelle, C pour Promela)
- Types abstraits algébriques

  (LOTOS, SDL => sémantique réécriture)
- Langages fonctionnels

  (LCS, E-LOTOS)

Evaluation des expressions

Cinq choix de conception "raisonnables"

1. Déterminisme des expressions
2. Absence d'effets de bord
3. Bonne initialisation des variables
4. Fonctions partielles => exceptions
5. Tolérer la non-terminaison éventuelle

Types de base

- Booléens, entiers, chaînes, etc.
- Ne pas figer la définition de ces types
  - nombres avec diverses précisions (matériel)
  - interprétation abstraite => changer la définition des types

```
module NATUREL is
  type NAT = {0, 1, MORE}
  function "+" (X, Y: NAT) is
        0 + X = X
        1 + 1 = MORE
        MORE + X = MORE
        Y + X = X + Y
  end
end
```

Types enregistrements ("struct")

Deux problèmes :

- Allocation/désallocation
- Bonne initialisation des champs

```
type POINT is struct (X, Y:int)
function F () : POINT is
    var P:POINT
    P.X := 0
    return P
end
```

Solutions :

- initialisations forcées : `var P : POINT := (0, 0)`
- affectations globales : `P := POINT (0, 0)`
- affectations champ par champ => analyse statique

Types tableaux

Trois problèmes :

- Allocation/désallocation
- Débordement d'indices
  - prévoir des levées d'exceptions
  - éviter les exceptions non pertinentes => analyse statique (polyèdres)
- Bonne initialisation des éléments

```
var X : array [10] of int
X [1] := 0
X [3] := X [2]
```

Solutions :

- initialisations forcées : `var X : array [10] of int := 0`
- affectations globales : `X [all] := 0`
- affectations champ par champ => analyse statique (polyèdres)

Types unions

Indispensables aux protocoles télécom. (multiplexage)

```
type REQUETE is
   union
      CONNEXION => ID:int, ADRESSE:int
      DECONNEXION => SESSION:int
   end
```

Quatre problèmes :

- Allocation/désallocation
- Bonne initialisation des champs (similaire enregistrements)
- Accès à un champ incompatible avec le discriminant => levées d'exceptions
- Accès au discriminant => risque de violation du typage

  Solutions : restreindre la manipulation du discriminant
  - obligation d'affectation globale (Ada)
  - instruction "case" avec filtrage (ML)

Types dynamiques

- Besoin : listes, arbres, files d'attente, etc.
- Deux approches antinomiques :
  - Langages à pointeurs/références explicites (C, C++, Ada, Java…)
  - Langages fonctionnels (ML)
- Les pointeurs posent des problèmes durs
  - sémantiques (déterminisme, concurrence)
  - efficacité de la vérification (model-checking)

  => ne pas distinguer deux cellules mémoire à des adresses différentes ayant le même contenu

Types dynamiques

- Si on veut des outils de vérification efficaces
  - pas de pointeurs explicites
  - types de données fonctionnels (ML)

- Il semble qu'il faille choisir entre :
  - l'approche à objets (notion de référence)
  - la vérification par model checking

Modélisation des comportements

Comportements séquentiels

Trois manières de décrire les automates

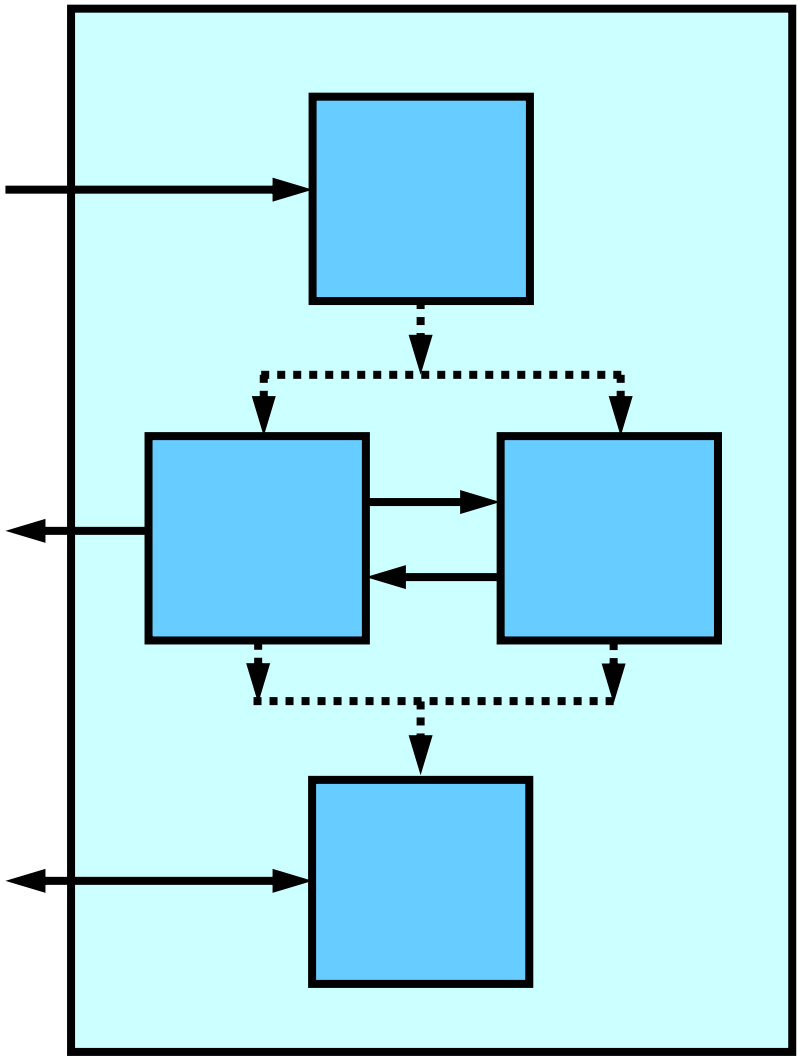
1. Formules logiques
2. Enumération états/transitions
   - Avantage : langage graphique
   - Inconvénients :
     - problème de la montée en complexité
     - maintenance difficile (programmation par "gotos")
     - modélisation des interruptions
     - coût élevé des outils graphiques
3. Expressions régulières

Comportements séquentiels

- Expression régulière =
  - événements atomiques
  - concaténation/séquence (noté ";" ou ".")
  - union/choix (noté "+" ou "[]")
  - répétition ("*", boucles "loop", récursion)
- Algèbres de processus = expressions régulières étendues avec
  - parallélisme
  - données (variables, expressions…)
  - éventuellement : temps quantitatif

Comportements parallèles

- Syntaxe textuelle
- Opérateurs algébriques
- Permet d'exprimer
  - mise en parallèle
  - imbrication de processus
  - masquage des canaux
  - mélange séquentiel/parallèle
- Plusieurs sémantiques
  - entrelacement
  - "vrai" parallélisme



Variables et flot des données

- Les processus ont une mémoire locale
- Règle 1 *(boîte noire)*

  l'accès à la mémoire locale et l'évaluation des expressions sont invisibles et atomiques
- Règle 2 *(bonne initialisation)*

  toute variable doit être affectée avant d'être lue

  Impact sur la composition séquentielle :
  - contraintes syntaxiques ("préfixage" CCS, CSP, LOTOS)
  - contraintes sémantiques (muCRL, E-LOTOS)

Indéterminisme

- Concept essentiel qui doit être présent

- AdP: opérateurs explicites pour l'indéterminisme
- Deux formes duales d'indéterminisme
  - Indéterminisme sur le contrôle

    A; P [] A; Q

    A; P [] τ; Q

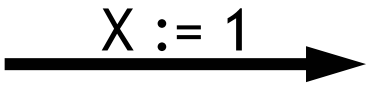
    τ; P [] τ; Q
  - Indéterminisme sur les données

    choice X:int [X <9]; P (X)

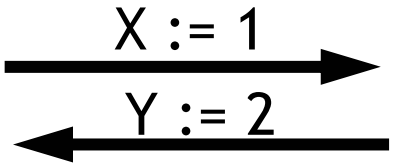
Rendez-vous

- Moyen unifié de synchronisation/communication
- Synchronisation : le premier arrivé attend l'autre
- Communication : émission (!), réception (?)

G !1 || G ?X



G !1 ; G ?Y || G ?X ; G !X+1



G !1 ?Y || G ?X !2

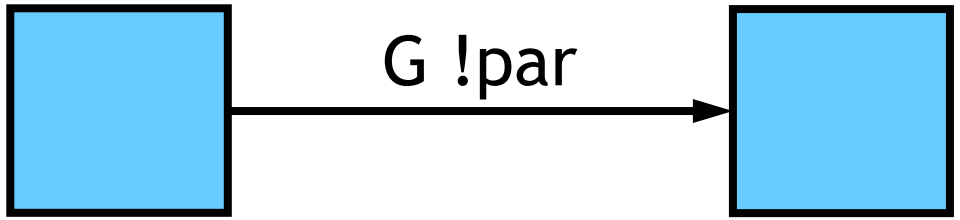
Avantages du rendez-vous

- Haut niveau d'abstraction
- Puissance d'expression
- Simplicité théorique
- Favorise la modularité
- Convient à plusieurs types d'implantation :
  - centralisées
  - distribuées
  - circuits

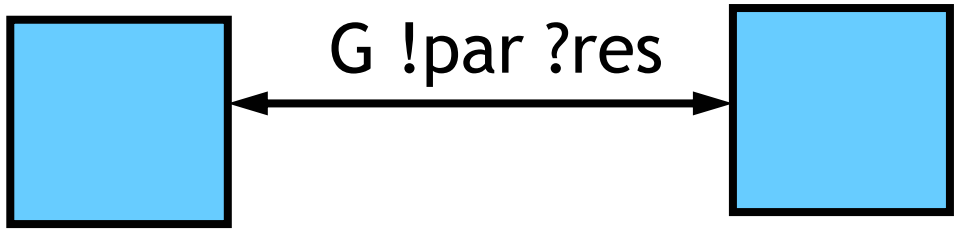
Evénements, appels de procédures

- Le rendez-vous permet de les exprimer

envoi d'événement



appel procédure (instantané)



appel procédure (non instantané)

Variables partagées

- Utilisation :

  Ada 95, Java, SDL, outils vérif. (Murphi, Promela, SMV, Uppaal)

- Points négatifs
  - suppose l'existence d'une mémoire centrale
  - distribution => problèmes sémantiques (JMM)
  - pas de sémantique unique (CRCW, *test and set*)
  - bas niveau d'abstraction
  - peu modulaire et peu compositionnel

- Choix des AdP
  - ne pas figer un modèle précis de variables partagées
  - les modéliser à l'aide du rendez-vous

Files d'attente

- Utilisation

  Langages de télécommunications (Estelle, SDL)

- Points négatifs
  - inaptes à certains domaines : matériel
  - file infinie : pas réaliste / file bornée : complexe
  - nombreux modèles de files

- Choix des AdP
  - ne pas figer un modèle précis de files d'attente
  - les modéliser à l'aide du rendez-vous

Rendez-vous avec filtrage

- On peut spécifier des conditions sur les valeurs reçues :

  G ?X [X > 1]

  G ?X ?Y [(X > 1) or (Y = 0)]

- Condition fausse => rendez-vous impossible

- Permet d'exprimer des hypothèses sur l'environnement

Rendez-vous avec négociation

- Deux processus peuvent se synchroniser sur deux réceptions

  G ?X [X > 1] || G ?Y [Y < 9]

  Effet:

  - choix indéterministe d'une valeur v dans]1, 9[
  - l'exécution continue après (X, Y) := (v, v)

- Spécification "par contraintes"
  - chaque processus exprime ses contraintes locales
  - le scheduler effectue l'intersection des contraintes
- Emission = cas particulier de réception

  G !V <=> G ?X [X = V]

Rendez-vous à *N* processus

- Synchronisation simultanée de *N* processus
- Intersection des contraintes
  - Vote: G ?X [P (X)] || G ?Y [Q (Y)] || G ?Z [R (Z)]
  - Diffusion: G !V || G ?X || G ?Y
- Généralisation du rendez-vous binaire
- Pouvoir d'expression irremplaçable
  - 5 exemples dans l'article
- Implantation
  - facile en contexte centralisé
  - plus délicate en contexte distribué

Conclusion

- Systèmes asynchrones : multiples applications
- Deux défis : complexité et correction
- De bons langages de modélisation sont essentiels
- Leur conception doit respecter 4 critères :
  - Expressivité
  - Universalité
  - Exécutabilité
  - Vérifiabilité

Conclusion

- Ces 4 critères admettent une solution :
  - Données => langages fonctionnels + analyse statique
  - Contrôle => algèbres de processus (temporisées)
- Remarque 1 : contour assez différent d'UML

  Double problème avec les objets :
  - sémantique : mélange objets + concurrence
  - efficacité : model checking et objets
- Remarque 2 : risque de conflit entre
  - goûts/envies des utilisateurs
  - contraintes sémantiques et technologiques

Algèbres de processus : le "plan B" ?

- Un corpus impressionnant de résultats :
  - modèles, langages, sémantique, équivalences
  - outils : compilation, génération de code, vérification
  - problème : transmission des connaissances

- Des applications industrielles en nombre croissant
  - CWI (Amsterdam), INRIA (Grenoble), Oxford…

- Exemple 1 : RNTL "Parfums" [article FMOODS 03]
  analyse d'un système Java (70 processus, 10^68 états) en moins de 20 minutes
- Exemple 2 : Bull : intégration de LOTOS/CADP au sein du cycle de développement

Pour en savoir plus…

http://www.inrialpes.fr/vasy