Modélisation des systèmes concurrents

Hubert Garavel

INRIA – Laboratoire d'Informatique de Grenoble

Université Grenoble Alpes, CNRS, Grenoble INP

http://convecs.inria.fr

Systèmes concurrents

- Définition : tout ce qui n'est pas séquentiel
- Cas des programmes :
  - programmes parallèles (ex: *threads* POSIX)
  - programmes distribués (ex: processus POSIX)
- Au-delà des programmes :
  - protocoles de télécommunications
  - circuits, processeurs, architectures matérielles
  - systèmes embarqués et cyberphysiques
  - organisations humaines (*workflows*)
  - flottes d'objets (IoT…) ou d'animaux (oiseaux…)

Difficultés intrinsèques

- Les systèmes concurrents sont complexes
- Problèmes fréquents :
  - interblocages (agents qui se bloquent mutuellement)
  - famines (agents indéfiniment bloqués par d'autres)
  - violations d'exclusion mutuelle, de contrôle d'accès…
- Raisons profondes :
  - non-déterminisme (chaque agent va à son rythme)
  - non-compositionnalité (les techniques de preuve pour les programmes séquentiels se transposent mal)

Détection des erreurs de conception

- Approches par simulation et test
  - basées sur le code exécutable (ou sur des modèles)
  - applicable à des systèmes complets
  - couverture incomplète : risque d'erreurs résiduelles
- Approches de vérification
  - basées sur des modèles formels "abstraits"
  - exploration exhaustive des états accessibles
  - *model checking* ou *equivalence checking*
  - multiples techniques pour réduire le nombre d'états

Langages de modélisation

- Caractéristiques attendues :
  - plus abstraits que les langages de programmation
  - sémantique formelle (définie mathématiquement)
- Diversité des langages proposés :
  - graphiques (automates, réseaux de Petri) ou textuels
  - parallélisme synchrone, asynchrone, temporisé…
  - sémantique d'entrelacement ou "vrai parallélisme"
  - communication par mémoire partagée ou par messages
  - propriétés sur les états ou sur les transitions (actions)
  - temps linéaire (traces) ou arborescent (arbres)

La saga des calculs de processus

GCL (*Guarded Command Language*)

- 1975 : E.W. Dijkstra invente les commandes gardées
- Instruction de choix avec gardes booléennes

  if $x \leq y \rightarrow$ min := x
  [] $x \geq y \rightarrow$ min := y
  fi

- Choix non-déterministe si gardes non exclusives

  if $x \leq y \rightarrow$ b := false
  [] $x \geq y \rightarrow$ b := true
  fi

  - avancée conceptuelle majeure
  - malgré une "résistance mentale considérable"

CSP (Communicating Sequential Processes)

- 1978 : C.A.R. Hoare propose le langage CSP
- Un modèle de calcul révolutionnaire :
  - processus concurrents sans mémoire partagée
  - communication par messages
  - mécanisme de rendez-vous qui unifie synchronisation et communication

    E !V || E ?X -- *la variable X reçoit la valeur V*
  - CSP reprend l'opérateur [] de choix non-déterministe
- Concepts clairs, mais pas de sémantique formelle

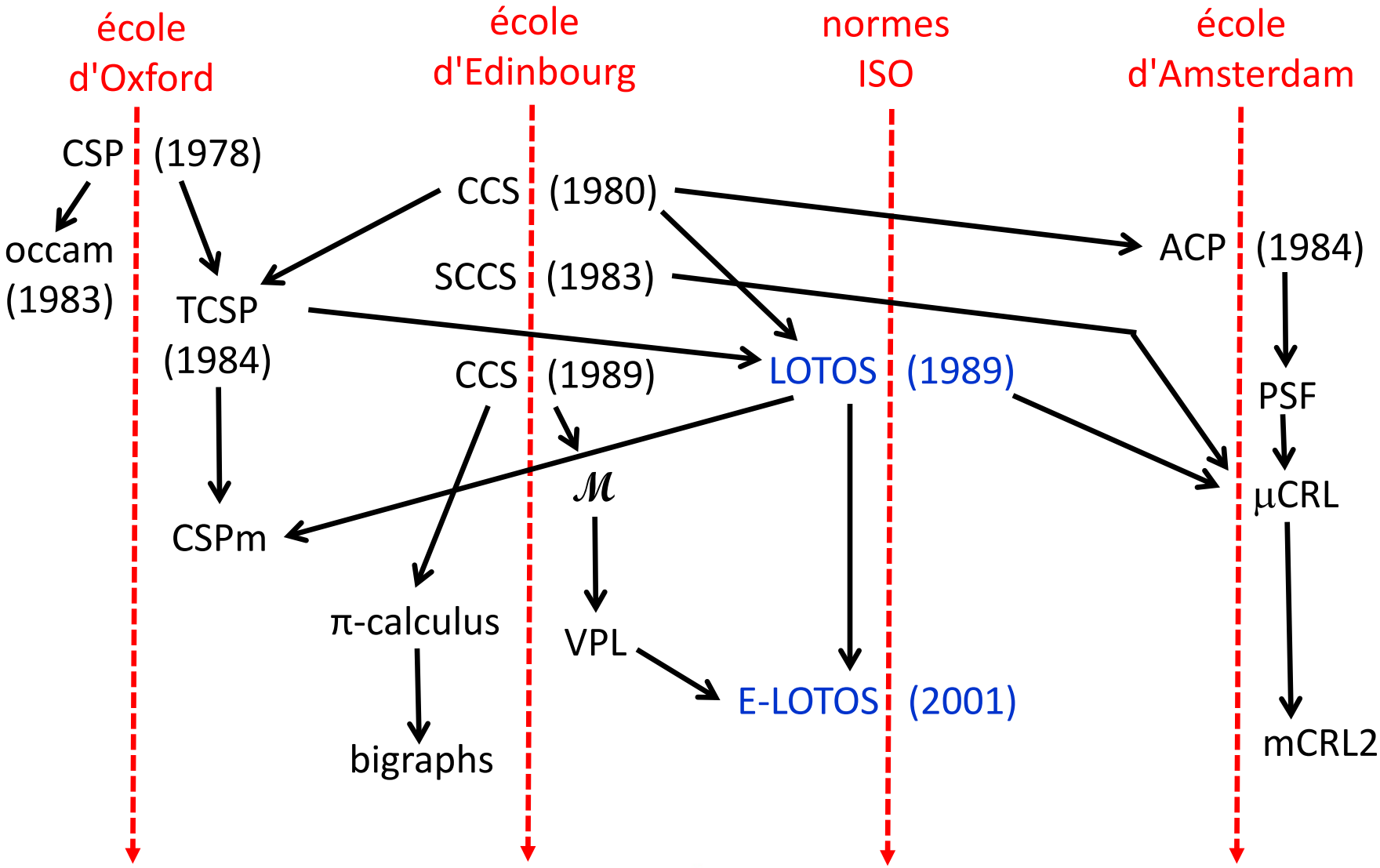
CCS (Calculus of Communicating Systems)

- 1980 : R. Milner propose le langage CCS
- Un langage minimal ("*calcul*")

$$P ::= 0 \mid a.P_1 \mid A \mid P_1 + P_2 \mid P_1|P_2 \mid P_1[b/a] \mid P_1 \backslash a$$

- Une sémantique formelle basée sur :
  - le modèle LTS (*Labelled Transition Systems*)
  - les règles SOS (*Structural Operational Semantics*)
  - les relations de bisimulations
- Une technique de preuve (equivalence checking) basée sur des transformations algébriques

Panorama des calculs de processus



Trois grandes "catastrophes"

- Java
  - parallélisme basé sur variables partagées et verrous
  - régression pré-Hoare
  - pas de sémantique formelle – JMM problématique
- UML
  - accent mis sur la syntaxe graphique
  - pas de sémantique formelle – vues incompatibles
- MDA-MDE *(Domain-Specific Languages)*
  - multiplicité des syntaxes – absence de sémantique

Après ces errements …

- Le besoin industriel de modélisation demeure
- La complexité des langages reste une barrière
- Ambition de l'équipe CONVECS :
  - fournir un langage simple, avec sémantique formelle
  - enraciné dans la théorie des calculs de processus
  - utilisable par des ingénieurs, sans long apprentissage
  - accompagné de compilateurs et outils de vérification
- Deux objectifs essentiels :
  - normalité : rester au plus près des langages usuels
  - lisibilité : être compris par des non-mathématiciens

Le langage LNT

(descendant d'E-LOTOS et d'OCCAM)

Spécifications LNT

- Chaque spécification est un ensemble de modules
- Chaque module contient :
  - des types
  - des canaux (pour typer les communications)
  - des fonctions (instructions pour calcul séquentiel)
  - des processus (comportements pour calcul parallèle)
- Fonctions et processus peuvent comporter :
  - des paramètres variables (**in**, **out**, ou **in out**)
  - des paramètres événements (exceptions, communications)
  - des préconditions (**require**) et postconditions (**ensure**)

Types LNT

- Trois classes de types :
  - Types prédéfinis : bool, nat, int, real, char, string
  - Types définis par des constructeurs libres :
    type POINT **is** NULL, POINT (X, Y: real) **end type**
    ce qui inclut les types énumérés, structures et unions
  - Types définis par des combinateurs variés :
    range 0..10 **of** nat, **array** [1..10] **of** bool, **list** of int,
    sorted list of int, **set of** real, X:real **where** X ≠ 0 …
- Typage fort, sans conversions implicites, inférence de type, ni ordre supérieur
- Surcharge possible des constructeurs et fonctions

Expressions LNT

- Propriétés sémantiques :
  - déterminisme, atomicité (évaluation en temps nul)
  - absence de communication, sauf levée d'exceptions
  - absence d'effet de bord (mémoire inchangée)
- Syntaxe des expressions :
  - variable ou accès à un tableau
  - appel (préfixé ou infixé) de constructeur ou fonction
  - accès à un champ de constructeur POINT.X
  - modification d'un constructeur POINT.{Y -> 0}
  - listes et ensembles en extension {1, 2, 3}

Instructions LNT

- Propriétés sémantiques :
  - déterminisme, atomicité (exécution en temps nul)
  - absence de communication, sauf levée d'exceptions
  - possibilité de modifier la mémoire locale
- Syntaxe des instructions :
  - **null**, déclaration de variable, affectation de variable,
  - **return**, **assert**, **raise**, composition séquentielle,
  - **if-then-else**, **case** avec *pattern-matching*,
  - boucles **for** et **while** avec instructions **break**,
  - appels de fonctions avec paramètres **in**, **out** et **in out**

Comportements LNT

- Propriétés sémantiques :
  - non-déterminisme, non-atomicité (temps non nul)
  - communication, synchronisation, interruption
  - possibilité de modifier la mémoire locale
- Syntaxe des comportements : idem instructions +
  - **stop**, communication avec entrées et/ou sorties,
  - choix non-déterministe, affectation non-déterministe,
  - boucle infinie (sans **break**), interruption, **trap**,
  - composition parallèle, déclaration d'événements,
  - appels de processus avec paramètres **in**, **out** et **in out**

Recours intensif à l'analyse statique

- Mission n° 1 : garantir la sémantique de LNT
  - "erreurs" destinées à rejeter des programmes illégaux
  - variables mal initialisées *(a ?x [] b ?y) ; c!(x+y)*
  - variables partagées *a ?x || b ?x*

  Seul LNT utilise l'analyse statique pour sa sémantique

  Les autres calculs de processus restreignent la syntaxe
- Mission n° 2 : prévenir les erreurs de l'utilisateur
  - "avertissements" concernant des programmes légaux
  - code mort, variables inutiles, affectations inutiles, boucles exécutées une seule fois, etc.

Implémentations de LNT

2 implémentations de LNT

- LNT2LOTOS : traducteur LNT → LOTOS
  - développé à la demande de Bull (2005)
  - permet de réutiliser les outils de vérification LOTOS
  - actuellement : 46 000 lignes de code (version 7.1)
- TRAIAN : compilateur LNT "séquentiel" → C
  - issu de la thèse de Mihaela Sighireanu (1999)
  - actuellement : 42 000 lignes de code (version 3.8)
- LNT2LOTOS et TRAIAN s'auto-compilent
  - 80% de leur code est écrit en LNT
  - et 20% en C et SYNTAX (générateur de compilateurs)

28 études de cas décrites en LNT

algorithmes distribués (6)	Google, Princeton, RWTH
avionique (3)	Airbus, IRT St Exupery
circuits et processeurs (6)	STMicro, Tiempo, Utah
cloud, fog et IoT (5)	Orange Labs, Rabat
interfaces homme-machine (2)	Atos, Toulouse
sécurité (3)	Graz, Laval
autres systèmes industriels (3)	Pise, Saarland

https://cadp.inria.fr/case-studies

14 outils générant du code LNT

AADL	Toulouse-Sfax
Applied π-calculus	Grenoble
BPEL-WSDL (2)	MIT-Tsinghua, Bucarest-Grenoble
BPMN (2)	Nantes, Paris
DFT	Twente
EB3	Paris-Grenoble
ESDL	Münster
GRL	Grenoble
Multi-agents	L'Aquila-Lucca
Objets IoT (2)	Grenoble-Nokia
UML-RT	Toulouse-Kingston

https://cadp.inria.fr/software

Conclusion

Conclusion

- Besoins :
  - Le parallélisme est indispensable, mais difficile
  - La vérification formelle est nécessaire
  - Pas de vérification sans langage(s) de modélisation
- Situation :
  - 1975-95 : enthousiasme pour les méthodes formelles
  - 1995-2010 : coup d'arrêt (erreur Java, utopie UML...)
  - Peu de langages réellement utilisables aujourd'hui
  - Peu d'initiatives pour créer de nouveaux langages

LNT : un langage moderne

- A mi-chemin entre programmation et modélisation
  - enraciné dans les calculs de processus : Hoare, Milner…
  - inspiré aussi des langages fonctionnels et impératifs
  - défensif : typage, pré/post-conditions, analyse statique…
- Simplicité d'apprentissage
  - apprécié des industriels : STMicroelectronics, Tiempo…
  - enseigné à Grenoble, CNAM Paris, Saclay…
- Implantations et applications
  - TRAIAN et LNT2LOTOS (~ 100 000 lignes de code)
  - 28 études de cas — 14 outils générant du code LNT

Remerciements

Mihaela Sighireanu

Guillaume Schaeffer

Lian Apostol

Alban Catry

Sai-Srikar Kasi

Jan Stoecker

TRAIAN 1.0 à 3.8

David Champelovier

Frédéric Lang

Wendelin Serwe

Xavier Clerc

Yves Guerte

Christine McKinty

Vincent Powazny

LNT2LOTOS 1.0 à 7.1